RDBG: a Reactive Programs Extensible Debugger

Erwan Jahier

To cite this version:


HAL Id: hal-01295229
https://hal.archives-ouvertes.fr/hal-01295229
Submitted on 30 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ABSTRACT

Debugging reactive programs requires to provide a lot of inputs – at each reaction step. Moreover, because a reactive system reacts to an environment it tries to control, providing realistic inputs can be hard. The same considerations apply for automatic testing. This work take advantage on previous work on automated testing of reactive programs that close this feedback loop.

This article demonstrates how to implement opportunistically such a debugging commands interpreter by taking advantage of an existing (ocaml) toplevel Read-Eval-Print Loop (REPL). Then it shows how a small kernel is enough to build a full-featured debugger with little effort. The given examples provide a tutorial for end-users that wish to write their own debugging primitives, fitting to their needs, or to tune existing ones.

An orthogonal contribution of this article is to present an efficient way to implement the debugger coroutining using continuations.

The Reactive programs DeBuGger (RDBG) prototype aims at being versatile and general enough to be able to deal with any reactive languages. We have experimented it on 2 synchronous programming: Lustre and Lutin.

Keywords

Programmable Debuggers; Dynamic Analysis; Monitor; Reactive systems; Synchronous languages; Interpreter; Compiler; Code Instrumentation; Continuations.

1. INTRODUCTION

Reactive systems and synchronous languages. A reactive system is an assembly of hardware and software that continuously interacts with its environment, typically via sensors and actuators. Because reactive systems are often critical, dedicated languages that offer strong guarantees on the software behavior were designed. Synchronous languages [1, 2, 11], for instance, ensure that the generated code uses a bounded amount of memory and execution time. A lot of efforts has also been put on formal verification and automated testing, but almost no work on debugging. Of course, for programs which by construction, contain no loop, no memory leak, and no segmentation fault, a debugger seems less necessary. Nonetheless, the remaining errors, revealed by testing or formal verification, can still be difficult to spot.

Debugging reactive programs: closing the feedback loop. One difficulty when debugging reactive programs is that they often require a lot of inputs – at each activation step. Moreover, those inputs are not always easy to provide, as they can depend on past values of the program inputs and outputs. Actually, we face exactly the same problem when performing automated testing: to provide realistic input sequences, we need to take into account the following feedback loop: the program influences its environment, which it turn influences (reacts to) the program. The testing tool of reactive programs LURETTE tackles this problem by providing a language dedicated to the modeling/simulation of program environments [12]. This language, named Lutin [24], is an extension of Lustre [11] designed to model non-deterministic reactive systems. To test automatically a program, this idea is therefore to write in Lutin the program environment model, and then to execute the program under test inside this simulated environment.

Since we have the same problem for debugging than for testing, the idea is to re-use the LURETTE infrastructure to close feedback loop. This is all the more natural to use the same framework that it is precisely the objective of a testing tool to detect bugs that a debugger would then help to track. In order words, to debug a reactive program, one can reuse the simulated environment used for the tests, which revealed the error that require the use of a debugger. Note that in this framework, the environment model is also a reactive program (written in Lutin), which therefore can also contain bugs, and also necessitates a debugger. Our debugger is actually able to debug Lutin programs too.

RDBG, an extensible debugger for reactive programs. In this article, we present RDBG, a debugger for reactive programs that re-use the LURETTE plumbing and its ability to plug various kinds of reactive programs together in closed loop. RDBG has a plugin mechanism to connect to reactive program runtime systems. Two language plugins are currently implemented: one for Lustre V6, and one for Lutin. RDBG is extensible in a second manner: it provides a language which allows programmers to write their own debugging and monitoring commands. This language is based on a kernel made of 2 primitives that allow one to inspect the debuggee enits, and to navigate through its runtime events list (observation points). The sequence of events comes from a pre-defined instrumentation of the target language runtime system that is provided via the language plugin.

The RDBG command interpreter. A debugger is a program that runs in coroutine with a program to be debugged, and that observes and controls its execution. It interacts with the debuggee by interpreting in loop commands, possibly coming from a Graphical User
Interface. For a debugger to be extensible, we need this commands interpreter to be able to interpret programs. An opportunistic way of obtaining such a commands and programs interpreter is to rely on an existing host language equipped with a good set of libraries, and an interactive Read-Eval-Print Loop (REPL); in this work, we use OCAML as the host language, which has both a REPL and a rich set of libraries. One drawback to rely on OCAML is of course that it requires to know OCAML to program debugging extensions. However, if one just wants to use RDBG, or maybe just wants to tune existing commands, knowing OCAML is not necessary. Likewise, the examples in this article are given in OCAML, but no deep knowledge of ml should be needed to understand them.

A continuation-based kernel. Finally, we propose an original and efficient way of implementing coroutine using continuations. Indeed, continuations allow to execute arbitrarily complex user code in debuggee process, which avoid costly context switches and communications between the 2 processes.

Debuggers and academic languages. Debuggers are fastidious to implement. From an academic perspective, working on debuggers is time-consuming and difficult to bring out. Nevertheless, debuggers are useful. This is the reason why:

- we have chosen to rely on existing REPL and libraries;
- we have paid a particular attention to re-usability by defining a language agnostic interface and by providing to compiler developers a simple plugin API;
- we have focused on separation of concerns by providing debugger users an API that allows them to implement their own debugging and monitoring programs with no need to look under the hood of the runtime system.

Another claim we try to argue in this article is that, once you have a carefully crafted programmable kernel, extending it with new debugging commands is easy and entertaining.

Plan. While RDBG aims at being versatile enough to debug any kind of reactive programs, it currently works with 2 synchronous languages. We therefore first briefly review them in Section 2 – albeit a deep understanding of synchronous languages is not necessary to grasp the debugging commands we show. Section 3 presents the RDBG user interface, and demonstrate how to build a full-featured debugger on top of a small kernel. Section 4 presents the RDBG plugin mechanism, that is the API for language developers that would wish to use it. Section 5 describes the design choices and the prototype implementation. Section 6 briefly discusses the 2 current language plugins. Section 7 presents related work.

2. SYNCHRONOUS LANGUAGES

RDBG is (currently) plugged onto 2 synchronous languages, Lustre and Lutin [16]. The examples we give in the forthcoming sections refer to Lutin, and even though a deep understanding of it is not necessary, we recap its main characteristics now. We first recap the Lustre main characteristics, as Lutin is based on Lustre concepts.

Lustre. A reactive system interacts continuously with its environment, at a speed imposed by the environment. Typically, it first acquires inputs via sensors, performs a computation step, and then sets its outputs through actuators. This cyclic behavior can be periodic (time-triggered) or sporadic (event-triggered).

Because reactive systems are often critical, dedicated languages that offer strong guarantees on the software behavior were designed. Synchronous languages [1, 2, 11] such as Lustre were successful in this respect as they make reasoning about time easier, thanks to the notions of logical time and deterministic concurrency. Synchronous languages generate code with known bounds on their memory usage and execution time.

The Lustre compiler (as well as Scade, its industrial version) generates a C step function, that is meant to be embedded in systems with no operating system. In Lustre, one defines reactive programs via sets of data-flow equations that are executed in parallel. Equations are structured into nodes. Nodes transform input sequences (or streams) into output sequences.

Lutin. In order to be able to test Lustre programs (or programs written in other reactive languages) automatically, we need to simulate the System Under Test (SUT) environment. To generate realistic input sequences, such simulated environment needs to take into account the SUT outputs. This simulated environment is therefore itself a reactive program which environment is the SUT. The Lutin language [24] was designed to program stochastic reactive programs, that can model reactive system environments [12, 14].

Lutin is a probabilistic extension of Lustre with an explicit control structure based on regular operators: sequence (fby, for “followed by”), Kleene star (loop), and choice (|). At each step, the Lutin interpreter (1) computes the set of reachable constraints, which depends on the current control state; (2) removes from it unsatisfiable constraints, which depends on the current data-state (input and memories); (3) draws a constraint among the satisfiable ones (control-level non-determinism); (4) draws a point in the solution set of the constraint (data-level non-determinism). This chosen point defines the output for the current reaction. The solver of the current Lutin interpreter uses Binary Decision Diagrams (BDD) and convex polyhedron libraries. It is thus able to deal with any combination of logical operators and linear constraints. Unlike Lustre, Lutin programs are not meant to be embedded, and only an interpreter is available.

2.1 A debugger-based Lutin tutorial

One benefit of a debugger is to allow programming language learners to discover its operational semantics. Maybe because language programming environments don’t always come with a debugger, language introductory tutorials are seldom based on them. Here, we propose to describe the semantics of Lutin on a trace produced the Lutin debugger we present later. The objective is actually less to present the semantics of Lutin (which is not the topic of the article) than to motivate the use of debuggers and to introduce some of our debugger concepts step by step.

Figure 1 contains a Lutin program made of two nodes, aut and env. The aut node produces a Boolean output b out of a integer input T, while env produces T out of b. As the names of those 2 nodes suggest, aut plays the role of the system under test (or program under debug), and env plays the role of its environment. Therefore
The Lutin semantics says that when there is no constraint, the output are generated at (pseudo-)random. Here the value \( f \) (false) is chosen.

At event 10 starts the second step. This time the elected constraint in \( \text{env} \) is \( "b \text{ and } T=pre T" \) (the \( \text{pre} \) operator gives access to the value of a variable at the previous step). The Lutin semantics states that when a constraint is elected, inputs and memories are replaced by their values (constant propagation). Hence, this expression is evaluated to \( "false \text{ and } T=4" \), which is equivalent to \( "false" \), which is unsatisfiable. This appears in the \( \text{usat} \) event 12.

Since this branch of the alternative failed to produce a solution, the other branch is tried (event 13): \( "not b \text{ and between}(T,5,10)" \) evaluates to \( "true \text{ and between}(T,5,10)" \), and then after expansion of the \( \text{between} \) macro to \( "5<T \text{ and } T<10" \). This constraint has 4 solutions, and event 15 shows that the random generator has chosen the value 6. For the \( \text{sut} \) node, it has chosen \( t \) (true) at event 19.

For the third step, the only satisfiable alternative for \( \text{env} \) is the first one (\( "b \text{ and } T=pre T" \)). By chance it is the one that is tried first (at event 22); after input and memory values propagation, it evaluates to \( "T=6" \), which has a unique solution. At step 4, there is no alternative, and the elected constraint is \( "T = 1 + pre T" \), which also has a unique solution (\( T=7 \)) (cf event 33). The trace events for step 5 are similar to the ones of step 2, except that the only satisfiable constraint is chosen in the first place this time.

Figure 2: An execution trace of the Lutin program of Figure 1

Each row in this trace corresponds to a particular observation point in the execution. The first event indicates that the \( \text{env} \) node is called first (which was specified to RDBG at some point). Events 2 to 4 inform that the constraint \( "T=4" \), is tried; obviously, this constraint is satisfiable and bounds the variable \( T \) to 4. Then comes the turn of \( \text{sut} \); events 4 to 8 inform that the elected constraint it "true", which is actually is the only one in this node. The Lutin semantics says that when there is no constraint, the output are generated at (pseudo-)random. Here the value \( f \) (false) is chosen.

In order to ease the work of eventual plugins contributors, and to encourage programmers to contribute to the RDBG libraries of debugging commands, the code has been licensed under GPL. Everything related to this free software project can be found at http://rdbg.forge.imag.fr/.

3. A REACTIVE PROGRAMS DEBUGGER

RDBG targets reactive languages. New languages can be added via a plugin mechanism described in Section 4. RDBG REPL relies on the OCAML one. RDBG is based on a small kernel, that can be extended by standard OCAML programs. In this section, we present this kernel (3.1), and how it can be adapted to various target languages (3.2). Then we demonstrate how it can be extended (3.3).

Figure 3: The chronograms of the trace of Figure 2

This execution is represented graphically in Figure 3 by chronograms representing the values taken by \( T \) and \( b \) at each step during the execution.

3.1 The RDBG User Interface

The RDBG User Interface is made of one data type \((\text{Event} . t)\) and one function \( \text{run} \), that starts the debugging session and returns the first event:

\[
\text{val run : params -> Event . t}
\]
where `params` holds the set of the debug session parameters (file names, number of steps to perform, interpreter options, etc.), and where `Event.t` is a structure defined as follows (in ml syntax):

```ml
type t = {
  name : string;
  lang : string;
  depth : int;
  nb : int;
  step : int;
  inputs : var list;
  outputs : var list;
  data : (string * Data.v) list;
  next : unit -> t;
  terminate : unit -> unit;
  sinfo : (unit -> src_info) option;
  kind : kind;
}
```

The field `name` holds the name of the node instance associated to the current event; `lang` holds its programming language name, and `depth` its depth in the call tree; `nb and step` respectively hold a counter that is incremented at each event, and at each step; `inputs` and `outputs` hold the node interface variable names and types; `data` holds the node variable values (interface and local variables). `Data.v` is a data type provided by RDBG that holds debuggee variable values (Boolean, integer, real, structure, array, enum); `next` holds a frozen function that, when called with the unit value, returns the next event; `terminate` holds a function that allows to terminate the execution cleanly; `kind` and `sinfo` hold the `src_info` and the kind data types, that are defined below.

```ml
type src_info = {
  expr : Expr.t ; (* current ctrl point *)
  atoms : src_info_atom list ;(* atoms expr is made of *)
}
```

The `src_info` data type holds source-level information. Its `expr` field holds an expression that represents the current control point; it explains how the outputs are computed from the inputs at the current step. The `Expr.t` data type is standard expression type that is supposed to be versatile enough to be able to represent all supported languages constructions. The `atoms` field holds a finer-grained representation of this expression; this information is split into a list of atoms, because the parallelism is a constitutive characteristic of synchronous languages, and thus the current control-point is often distributed in all over the source code.

```ml
type src_info_atom = {
  expr : Expr.t ; (* current ctrl point *)
  line : int ; (* line nb begin/end in file *)
  char : int ; (* char nb begin/end in file *)
  stack : src_info_atom list ; (* call stack *)
}
```

The `src_info_atom` is a structure made of a file name (`file`), the beginning/ending character and line numbers of the atom in the source code (`char` and `line` fields). The `stack` field holds the call stack.

```ml
type kind = Top | Call | Exit | MicroStep of string
```

The `kind` data type is used to classify the various kinds of events. `Top` events are generated at the top-level of the cyclic process, and before a new round of reactive program steps is started (it is therefore where the `step` counter is incremented). `Call` events are generated when a node is called, and `Exit` events when it is exited. `MicroStep` events are generated during the execution of the node.

The various kinds of `MicroStep` depend on the node language. We give examples of possible `MicroStep` kind sets for the Lutin language below (3.2).

This interface aims at being general and versatile enough to represent runtime information of any reactive languages. At least it is able to represent the information of Lustre and Lutin. Now we present a few Lutin specificities.

### 3.2 The Lutin Micro-steps

The `MicroStep` variant of the event kind data type of Section 3.1 is meant to hold language specific event kinds. Choosing a good set of event kinds is a delicate task: it defines the granularity of the instrumentation, and is the result of a trade-off between precision and efficiency. Too many events would slow-down the execution, but missing events could prevent to understand what’s going on with a buggy program. Discussing the current instrumentation granularity we made for Lutin is out of the scope of this article, which aims at presenting the debugger language. However, we have chosen to briefly present it because we use them in some examples we give in Section 3.3. Moreover, it illustrates the RDBG API versatility.

In order to report faithfully what happens during a Lutin program execution, stopping at call or exit time is not enough. We need a finer-grained observation of the execution, that indicates which expressions contribute to the computation of the outputs of the current step. Lutin expressions are made of constraints that may or may not be satisfiable, depending on the inputs, and on the past (memory values). Hence we have defined, specifically for the Lutin plugin, 3 new kinds of events: `try` events, when a constraint is elected; `sat` events, when the selected constraint is satisfiable and (hence) is used to compute the outputs; `usat` events, when the selected constraint is not satisfiable.

```ml
type lut_evt =
  | Top | Call | Exit
  | Try (* Try a constraint to compute the step *)
  | Sat (* The tried constraint is satisfable *)
  | Usat (* The tried constraint is Not satisfable *)
val to_lut_evt : Event.kind -> lut_evt
val from_lut_evt : lut_evt -> Event.kind
```

### 3.3 What can be done with this small kernel

We now demonstrate by examples how such a minimalist kernel can be used to implement classic and advanced debugging features. Some of them might not be the more efficient solution, but the longest is about 20 lines long. Indeed, the emphasis here is put on readability and conciseness, rather than on efficiency. For example, each time we use linear-time lists, logarithmic-time association tables (a.k.a. maps) would have been more efficient but less readable.

All programs described below are part of the standard RDBG library module RdbgStdLib, available in the RDBG git repository.

#### 3.3.1 Forward Moves

Moving to the next event (`next`) just requires to unfreeze the `next` field function of the current event. Moving several events forward (`nexti`) can be done using a counter (`i`) that is decremented until it becomes equal to 0:

```ml
let rec (nexti : e -> int -> e) =
  fun e i -> if i > 0 then nexti (next e) (i-1) else e
```

Similarly, we can implement commands that go to a specific event number (goto_i), or step number (goto_si):

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

Note that here and in the following, we present purely functional commands. Equivalent commands using side-effects are easy to define (and sometimes easier to use) like this:

```
let stop_here e =  
  if e.nb < 42 then goto_i e 42 else e
```

This is now a reference to an event. `ni` only takes as argument the number of forward steps to perform, and modifies the event reference through a side-effect.

### 3.3.2 Conditional Breakpoints

The next_cond command below takes as argument a predicate (cond) over events which states when to stop moving forward.

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

The criterion to decide when to stop going forward can of course be arbitrarily complex. The only limitation on what can be done comes from the information contained in the event itself. For example, suppose you notice that your program violates some invariant (or test oracle); one can then move forward up to the first event where the invariant is violated to inspect the state of your program. More specifically, suppose that this invariant consists of an alarm that should be set to true when a numeric variable “T” overcomes a threshold. One could write the following interactive query at the RDBG prompt:

```
(rdbg) let e = ref ( run params );;  
(rdbg) ni 42 ;;  
(* moves 42 events forward *)
```

The invariant, encoded into the `stop_here` predicate, is then provided to the `next_cond` command. `vi` and `vb` (which code is straightforward and part of RdbgStLlib) respectively extract from the data field of an event the value of an integer and a Boolean variable.

### 3.3.3 Forward Moves (cont)

As we have mentioned before, several events can have the same step number. When navigating into the trace history of cyclic data, it is sometimes interesting to go to the next step of the current node. To implement such a command, we can again take advantage of the `next_cond` command:

```
let rec (step : e -> e) =  
  fun e -> next_cond e
  (fun ne -> e.name = ne.name && e.depth = ne.depth &&  
   e.kind = ne.kind && e.step = ne.step-1)
```

We can define a `stepi` command using `step` in exactly the same manner we have defined `nexti` using `next`. By using this `step` command instead of `next` in the definition of `goto_si` (Section 3.3.1), we obtain a new behavior that some users might prefer over the initial one. It is the whole purpose of a programmable debugger to provide a tool that makes it easy to fit one’s specific needs or preferences.

### 3.3.4 Hooks

Hooks are another very useful mechanism to let users tune their environment. They can be implemented, for example using hash tables, as follows:

```
let (hooks : (string, (Event.t -> unit)) Hashtbl.t) =  
  let ht = Hashtbl.create 1 in  
  Hashtbl.add ht "print_event" print_event; ht
  let add_hook = Hashtbl.replace hooks
  let del_hook = Hashtbl.remove hooks
```

Hook functions are stored into a table that maps a string identifier to functions over events. This string can be used to remove or update the associated function. Here, we initialize the hooks association table with a single hook that contains an event printer. The idea then is to modify the `next` function of Section 3.3.1 and apply hook functions at each event.

```
let rec (next : e -> e) =  
  fun e ->  
    let me = e.next () in  
    Hashtbl.iter (fun _ f -> f me) hooks; me
```

The trace of Figure 2 in Section 2.1 has been produced calling in loop this `next` function.

### 3.3.5 Custom Traces

Depending of the debugging context, customizing the trace that is printed by the debugger is useful. For instance, not all events are of interest, and for events that are printed, different information might give more insights. Suppose for example that one wants to modify the trace of Figure 2 in two manners. Firstly, by focusing on events that occur in the env node (indeed the nut ones are boring). Secondly, by printing constraints expressions where input and memory values are propagated. To do that, one can get the expression (Expr.t) contained in the exps field of the sinfo field, as well as the variable values in the data field, and program an expression pretty-printer that instantiates bound variables.

```
(rdbg) let (my_pp_event : e -> unit) = fun e ->  
  if e.name = "nut" then ()  
  else sinfo2str e  
  let cstr = match LutinRdbg.to_lut_evt e.kind with  
    | Top | Call | Exit -> ""  
    | Try -> (" \%"+(sinfo2str e) "\%")  
    | Sat | Unat ->  
      (match e.sinfo with None->"" | Some si ->  
        "\%"+(Expr.pp si.expr.e.data) "\%")  
    in  
    Printf.printf "%Zi %Zi %s %s (%s) e.nb e.step (kind2str e.kind) e.name (data2str e.data) cstr;  
    flush stdout ;
```

In addition to the expression printer, this event printer uses 4 straightforward functions that we don’t show the code of either: to_lut_evt, that translates MicroStep event kinds into Lutin event (lut_evt) for type safety; sinfo2str, kind2str, and data2str, are used by the default event printer, translate sinfo, kind, and Data.v to strings. Once the new printer is defined, one just needs to update the `print_event` hook, as illustrated in the RDBG interactive session below:

```
(rdbg) del_hook "print_event";;  
(rdbg) add_hook "print_event" my_pp_event;;  
(rdbg) let e = stepi (run()) 4;;
```
The last line which calls the `stepi` command of Section 3.3.3 produces the following trace:

```
1 1 call env []
2 1 try env [] "T = 4"
3 1 sat env [] "T = 4"
4 1 exit env [T=4]
5 1 top - [T=4,b=f]
6 2 call env [b=f]
7 2 try env [b=f] "b and T = pre T"
8 2sat env [b=f] "false and T=6"
9 2 try env [b=f] "not b and between(T,5,10)"
10 2 sat env [b=f] "true and between(T,5,10)"
11 2 exit env [b=f,T=6]
12 2 top - [T=6,b=t]
13 3 call env [b=t]
14 3 try env [b=t] "b and T = pre T"
15 3 sat env [b=t] "true and T = 6"
16 3 exit env [b=t,T=6]
17 3 top - [T=6,b=t]
18 3 call env [b=t]
19 3 try env [b=t] "T = 1 + pre T"
20 3 sat env [b=t] "T = 1 + 6"
21 3 exit env [b=t,T=7]
22 3 top - [T=7,b=f]
23 3 call env [b=f]
24 3 try env [b=f] "not b and between(T,5,10)"
25 3 sat env [b=f] "true and between(T,5,10)"
26 3 exit env [b=f,T=9]
27 3 top - [T=9,b=f]
```

This is trace of exactly the same execution as the one that generated the trace of Figure 2. To obtain the same execution, one needs to provide the same seed the Lutin pseudo-random generator. Such expression printers or evaluators are good examples of code that can be written once for all, leveraging the work of plugins writers.

### 3.3.6 gdb-like Breakpoints

Another classic feature of debuggers consists in moving forward in the event history by setting breakpoints. We show a possible implementation of breakpoints that uses a syntax reminiscent to the gdb one. Basically, one can set a breakpoint on a node (break "a_node_id") or on a particular line of a particular file (break "[file::line][file::line]". We store each breakpoint in a table indexed by a global counter `bc`. This index can then be used to remove a specific breakpoint:

```
let ( profiler : bool -> unit ) = fun on ->
    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 stop e = List.exists (brk_matches e) ! brkpts in
let (continue : e -> e) = fun e ->
    let stop e = List.exists (brk_matches e) ! brkpts in
next_cond e stop
```

Now, the only non-trivial part consists in digging into the event data structure for the file name and line number. This work is done by the `brk_matches` event predicate below, that returns true if and only if the event argument matches the breakpoint string. This time the code is a bit laborious. Nevertheless, we report it here to justify the fact it fits into a few lines of code.

```
let bc = ref 0
let (brkpts: (int * string) list ref ) = ref []
let (break : string -> unit )=
    fun str ->
        let (bc,str):=(brkpts):=brkpts; incr bc
let (del: int -> unit ) =
    fun i ->
        brkpts := List.remove_assoc i
let (del_all:unit->unit ) = fun () -> brkpts:=[]; bc:=0
```

Readers that want to check that code, and that are non-familiar with the `Str` module of the OCAML standard library, just need to know that a call to `Str.split (Str.regexp ":\s:\")` on the string ""fn::42"" returns the list of strings [""fn"",""42""]. Notice here the advantage of having a rich existing library to ease the engineering a set of debugging commands on top of this small kernel. Now, implementing the (classic) `continue` command that moves forward until a breakpoint is reached is a simple as:

```
let (continue : e -> e) = fun e ->
    if on then add_hook "profile" prof_add
else del_hook "profile"
```

We could refine this command and take into account the character numbers. We could also have a better handling of errors in breakpoint syntax.

#### 3.3.7 Profilers

Now we illustrate how to implement a simple counting profiler. For purely data-flow languages like Lustre, such a profiler is completely useless, as every equation is used at each step. Therefore we present now a command that work for Lutin. It is specific to Lutin as it uses the micro-events presented in Section 3.2. The idea is simply to stop at `try` events and to associate in a table constraints and counters. Each counter is initialized to 1 when a constraint is elected for the first time, and incremented the following times.

```
let (prof_add : e -> unit ) = fun e ->
    try let cpt = Hashtbl.find prof_tbl tbl in
        Hashtbl.replace prof_tbl tbl (cpt+1)
    with Not_found -> Hashtbl.add prof_tbl tbl 1
let stop e = List.exists (brk_matches e) ! brkpts in
let (continue : e -> e) = fun e ->
    let to_lut_evt e.kind , e.sinfo with
        Try , Some si -> List.iter incr_prof (si ()).atoms
    in
```

This profiler could be enhanced (using `sat` and `usat` events) by computing 2 counters per constraint, to count the number of times an elected constraint is satisfiable.

Then we can rely again on the hook mechanism of Section 3.3.4 to implement a command that sets on and off the profiler mode:

```
let (prof_add : e -> unit ) = fun e ->
    if on then add_hook "profile" prof_add
else del_hook "profile"
```

At this stage, the only remaining work is to write a `prof_tbl` pretty-printer. One could also implement a coarse-grained time-profiling using the Unix.time() function at `call` and `exit` events for instance.

### 3.3.8 Backward Moves (Time-travel)

Efficient reversible debuggers [9, 28] store periodically (at `exec_rate` rate) events in a table. Then, in order to move backward, it suffices to move to the lastly stored event, and then to move forward until an event that satisfies the event predicate is found, as illustrated by the arrows labeled by 1 in the drawing below.

```
let (brk_matches : e -> string -> bool ) =
    fun e b ->
        match e.sinfo, Str.split (Str.regexp ":\s:\") b with
        | _ -> false (* no more breakpoint *)
        | Some _ -> false (* no src info at current evt *)
        | Some _ , [node] -> e.name = node
        | Some src, [file::line] ->
            let i, atoms = int_of_string line,(src ()).atoms in
            List.exists (fun { line=(debut,fin) ; file=f } ->
                f=file && debut <= i && i <= fin) atoms
        | _ -> failwith "syntax error in brkpts"
```
When no satisfying event is found, we start again with the previously stored event (cf the arrows 2, 3, and 4). When a satisfying event is found, we still need to go forward up to the last backward jump source, to make sure that no more recent event satisfies the event predicate (cf the 2 bullets representing satisfiable events, and the arrow 5). Of course this only works with programs with no side-effect.

```ocaml
val rev_cond : e -> (e -> bool) -> e
let rec rev_aux e i = (* Search for ce in ]e.nb;i[ s.t. p ce holds *)
  let e' = if e then e
  else next_cond e (fun e -> p e|e.nb=1)
  in
  if p e then
    rev_aux_last e e i (* search for a more recent *)
  else (* e.nb = i *)
  let x = (e.nb / !ckpt_rate - 1) in
  if x < 0 then
    (print_string "No suitable event found.\n"; Event.set_nb 1;
     Hashtbl.find ckpt_tbl 0)
  else
    let e = Hashtbl.find ckpt_tbl x in
    Event.set_nb e nb;
    rev_aux e ((x+1) * !ckpt_rate -1)
    and rev_aux_last e e_good i = (* Search for e in ]e_good;i[ s.t. p e holds *)
    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
    rev_aux_last e e_good i
  in
  if e.nb = 1 then
    failwith "Cannot move backward from event 1."
  else
    let x = ((e.nb - 1) / !ckpt_rate) in
    let last_e = Hashtbl.find ckpt_tbl x in
    Event.set_nb last_e nb;
    rev_aux_last e (e.nb-1)
```

3.3.9 Dynamic code modifications

A simple thing we could do to allow dynamic code modification from RDBG would be to change slightly the type of the next field of Event.t, and replace the unit type by an event. Although it might looks a bit dangerous, such kinds of features can, in some circumstances, be useful, for instance to test quickly (without modifying the source code) an hypothesis over the debuggee behavior.

4. THE PLUGIN API

We now present the plugin Application Programming Interface, that is the API for developers that would wish to plug their runtime system (compiler or interpreter) to RDBG. In order to do so, one needs to provide an OCaml library that implements the following interface:

```ocaml
type sl = (string * Data.v) list (* substitutions *)
type e = Event.t (* a useful alias *)
type plugin = {
  type e = Event.t (* a useful alias *)
  inputs : (string * string) list; (* name and type *)
  outputs : (string * string) list; (* name and type *)
  init_inputs : sl;
  init_outputs : sl;
  step : sl -> sl;
  step_dbg : sl -> e -> (sl -> e -> e) -> e;
}
val make : string array -> plugin
```

The sl alias type (substitution list) is used to hold variables instantiation. The plugin data type holds the list of interface variable names and types in its inputs and outputs fields. It also contains two fields, init_inputs, and init_outputs, to provide initial values to interface variables (i.e., the value of their memories before the first step). Then there are 2 step functions. One simple step function computes an output substitution out of an input one; this function is used to run program as LURETTE would. RDBG can then be used as testing tool. The step_dbg function is more complex, and holds the instrumented version of the runtime system; step_dbg exposes its depths to RDBG via events. Its first parameter is an input variable substitution (the same as step). The second parameter is the event of the current step (the initial event is built by RDBG). The third parameter is the continuation function. It holds the code that has to be executed after; it is used to fill the next field of Event.t. Notice that the output substitution is not returned by step_dbg, but it should be used in the continuation function instead.

In order to be used with the RDBG interactive command interpreter, plugin modules implementing this interface must be provided under the form of a byte-code library (.cma). RDBG commands can also be executed in batch using ocaml native compiler. In that case one also has to provide a dynamic native code library (.cmxs) version of the plugin.

We have currently implemented 4 such plugins. We have already mentioned the 2 language plugins for Lustre (LutinRun) and for Lustre (Lust2licRun). We also have an OCAML plugin (OCamlRun) that allows to plug programs that are executed via a compiler that uses ocaml as a back-end (cf the examples/ocaml/ directory of the RDBG git repository). We also have a plugin based on standard input and standard output reading and writing (StdioRun). This one is useful to enter inputs manually, via the keyboard or a GUI. It can also be used to plug any kind of systems via TCP/IP sockets. Of course this plugin provides no step_dbg function.

A few additional details about the Lustre and the Lutin plugins are given in Section 6.

5. COROUTINE VIA CONTINUATIONS

The debugger works in coroutine with the debuggee, and navigates through the event sequence generated by the program execution. Alternative manners of implementing coroutine are discussed in Section 5.4. In RDBG, as the plugin interface suggests, the coroutine relies on continuations. The idea is the following. Transforming a function into Continuation Passing Style (CPS) leads to a function that never returns. It never returns because each function is supposed to call, at its last breath, another function (its continuation), that itself never returns. Once all functions are in CPS, instead of calling the continuation, we can return a data structure (Event.t) that contains a frozen function, which allows to return the control back to the calling function (next field). This calling function (the debugger) hence runs in coroutine with the continuation function (the debuggee). Using this technique, it is easy to execute user code in the debuggee process, which avoids costly context switches and inter-process communications.

In the following, we illustrate this idea by describing how we have transformed the LURETTE top-level command interpreter to implement RDBG.

5.1 From LURETTE to RDBG via CPS

We illustrate the old idea of transforming a function into CPS on a vanilla LURETTE top-level function. Once all the administrative work has been done (parsing the various test parameters such as the test length, the name of the SUT, the name of its environment,
the name of the file where to save generated data, etc.), LURETTE
basically performs: (1) a step of the environment; (2) a step of the
SUT which LURETTE provides the environment outputs of the
previous step; (3) a step of the environment to which LURETTE
provides the SUT outputs of the previous step; (4) a step of the test
oracle, that checks if some property is violated; (5) a test to check
if the required number of steps is reached, and if it is not the case,
loop back to item 2.

Here is a Vanilla implementation of the LURETTE process, with a
SUT that increments its input by one, and an environment that
increments its input by 2, and this during 42 steps (we skip the
oracle step).

```ocaml
let ( step_sl : 's list -> ('s list -> 's list) list ->
fun sl sut_step_sl_l ->
List . flatten (List . map (fun f -> f sl) sut_step_sl_l) ) =
match stepl with
| [] -> cont ctx (List . flatten (res_stepl))
| step::stepl ->
step sl ctx (fun res_sl ctx ->
iter_step stepl (ctx,(res_sl::res_stepl))
) in
iter_step step_dbg_sl_l (ctx,[[]]) sl
```

Now we need a CPS version of it. Here is a possible solution
that works (and that is used by RDBG):

```ocaml
let ( step_dbg_sl : ('s list -> 'e -> ('s list -> 'e) list ->
's list list -> 'e -> ('e -> 's list list -> 'e) list -> 'e) ) =
fun step_dbg_sl_l sl e cont =
let rec iter_step stepl (ctx,res_stepl) sl =
match stepl with
| [] -> cont ctx (List . flatten (res_stepl))
| step::stepl ->
step sl ctx (fun res_sl ctx ->
iter_step stepl (ctx,(res_sl::res_stepl))
) in
iter_step step_dbg_sl_l (ctx,[[]]) sl
```

### 5.2 Instrument the runtime system with events

For the sake of simplicity, we consider here a simplified event
type. Doing the same on the event type defined in Section 3.1 is
not more difficult.

```
type event = { msg : string ; next : unit -> event }
```

Once the LURETTE interpreter is in CPS, instrumenting it with an
event is straightforward. Here is our CPS Vanilla LURETTE top-
level instrumented with events:

```ocaml
let plus a b cont = {
msg = Printf.printf "%i+%i\n" a b ;
next = fun () -> plus a+b cont
}
```

Readers of the pdf version of this article are encouraged to try
to copy/paste this code out into an OCAML top-level. Then, for
example, to launch a 42-steps test session, one just need to do this:

```ocaml
let e = ref (loop 42 0) in while(true)
do print_string ("e.msg"::"a");e::e.next() done ;
```

### 5.3 Handling lists of steps

Most of the modifications in the LURETTE code to transform
it into CPS were as straightforward as what is described in
Section 5.1. The only convoluted one is the following.

The real implementation of LURETTE is able to run concurrently
several programs under test, using several environments (and sev-
eral oracles to assess the test result). LURETTE therefore iterates
over a list of step functions, to compute a list of outputs from a list
of inputs. More precisely, we need a function that take as input 2
parameters : a list of variable substitution (holding the global step
input values), a list of step functions, that generate some output val-
ues. And this function computes from this the substitution holding
the global step output values. Here is a possible ml implementation
of such a function.

```
let ( step_sl : 's list -> ('s list -> 's list) list ->
fun sl sut_step_sl_l ->
List . flatten (List . map (fun f -> f sl) sut_step_sl_l) ) =
```

5.4 Other debugger technologies

One contribution of this article is to base the coroutine between
the debugger and the debuggee on the use of continuations. Hence
we review here what other debuggers do in this respect.

**Code Instrumentation.** In order to implement debuggers, some
code instrumentation is required for the coroutine with the de-
buggee. Typically, a test is added before each event site to decide
whether the control should remain in the debuggee, or should go
into the debugger. This test can be added via a source to source trans-
formation [8, 28], or at compile time via a compiler option that
will add those tests in the generated code. The granularity of this
instrumentation influences the performance.

Another option is to modify the executable program at run time
(e.g., using ptrace) and to add instructions to jump into the debug-
ger [27]. The advantage is of course at the performance side, as one
doesn’t need to perform additional tests at all. But of course such
debuggers depend on the target platform. This (mainstream) family
of debuggers even often exploits dedicated processor instructions
deal with memory modifications, which is particularly important
for low-level languages with no automatic memory management.

Some debuggers (e.g., Caml and Java) adopt a intermediate so-
lution, which consists in modifying the byte-code on the file. The
resulting debuggers remain platform independent, and save tests.
But going though a byte-code interpreter has a performance impact.
Another (minor) disadvantage is that the native and the byte-code
backends might sometimes behaves differently.

Another very popular way of designing debuggers is to instru-
ment a source code interpreter (as we do here). In that case the
tests are embedded in the interpretation algorithm. Is is certainly
the easiest method to implement debuggers when an interpreter ex-
ists. But developing interpreters on purpose duplicates the develop-
ment and the maintenance effort when a compiler exists. Moreover,
such interpreters generally run slower than the compiler equivalent,
and may behave differently.

In the Lustre RDBG plugin (Lutin has no compiler), we mini-
imize that problem as the instrumented interpreter operates over a
data structure obtained at the very last pass of the compiler: the
one that is pretty-printed to perform the code generation. Errors in
this pretty-printing would lead to bugs that would be hided in the
debugger indeed. An alternative would have been to instrument an
OCAML Lustre code generator.
Coroutine. Most debuggers live in a separate process, and communicate with debugees through sockets, pipes, or signals. The advantages of having two distinct processes are: (1) to be more robust to memory leaks, and program crashes; (2) to have a better separation of concerns; (3) to allow remote debugging; (4) to allow to debug several programs at the same time.

On the other hand, having the 2 routines in the same process (1) permits a direct access to the runtime system depths; (2) avoids costly context switches (which are less costly than using threads); (3) avoids costly messages writing/parsing (for sockets and pipes).

6. ABOUT THE CURRENT PLUGINS

Describing in detail how we have implemented the Lustre and the Lutin RDBG Plugins is out of the scope of this article. A few remarks still.

Plugins Design. The step_out and the step_env functions of Section 5.2 typically (but not necessarily) comes from a Lustre and a Lutin program respectively. In the corresponding plugins, step functions have been transformed into CPS and instrumented in a similar manner. The fact that OCAML is the language used to program LURETTE/RDBG as well as the Lutin and the Lustre interpreters makes the work easier, when it comes to provide an OCAML library that implement the RDBG plugin interface.

The important and difficult work is actually to define a good set of event kinds, so that the operational semantics of the underlying language is usefully depicted by the generated events. Some information can be reconstructed inside the RDBG process, but not all. Nevertheless, it is always possible to control the granularity of the instrumentation when compiling the plugin, and use different granularity for different runtime analysis.

For Lutin, the choices we made with respect to event granularity are described in Section 3.2. For Lustre, our current RDBG-plugin prototype has no micro-step event; each equation generates at most 2 events. It generates a call and an exit event if the equation is a node call, and it generates no event if for variable assignments (wires). This choice only makes sense because the Lustre V6 compiler [15] does not inline all the nodes (unlike the previous versions of Lustre compilers).

Plugins Performance. We have performed a few timing measurements on a simple Lutin program that performs 4 node calls made of trivial constraints (equalities). Then we have compared the execution time of that node executed with the step function, and with the step_dbg function executed via the RDBG command below:

```
let rec loop e = loop (e.next()) in loop(run());
```

and have observed a penalty smaller than 1%. When executed with the profiler of Section 3.3.7, the slowdown is of 30%. A Lutin node with more complex constraints would exhibit a negligible slowdown, as the constraint resolution would dominate the whole execution time. To give an order of idea, the number of events that are generated per second is around 4000 (on a PC running at 3.6GHz, because equations are easier to compute than constraints. More systematic benchmarks should definitely be done on the language plugins.

Note that actually, one source of inefficiency is due to the use of the ocaml top-level, which implies the use of the byte-code compiler. Indeed, the Lustin interpreter is compiled with the ocaml native code compiler (ocamlopt). The performance ratio between the byte-code and the native code compiler is reputed to be of 5 in average (it was 2 on our small program). Nevertheless, this drawback only concerns interactive sessions. It is still possible to run RDBG scripts in batch mode, and to benefit from the native code compiler performances. For efficient interactive sessions, a solution would be to use a native code top-level [10].

7. RELATED WORK

Debugging reactive systems. Ludic is a debugger of Lustre programs [19]. It features standard step-by-step backward and forward execution and conditional breakpoints using a dedicated Lustre interpreter. It also provides an algorithmic debugging engine. Algorithmic debugging consists in exploring a tree (here, a dataflow network) by asking yes/no questions to the user until the bug is found [26]. Implementing on top of RDBG the Ludic exploration algorithm, that fits into one page, should be straightforward. The advantage would be to take advantage of the modular compilation capabilities of the Lustre V6 compiler. Indeed, the embedded Ludic Lustre interpreter inlines the whole Lustre program, and only the top-level node is observable. The Lustre V6 compiler generates “synchronous objects” that are either interpreted (by the RDBG plugin or the Lustre V6 interpreter) or pretty-printed into a C function. The questions asked by the exploration engine might thus be much more precise and easier to answer than it is on compiled code.

Sabry and Sparud performs dataflow reactive programming on top of Haskell using streams and lazy computations. They designed a debugger [25] that gives insights on the behavior of such programs at the level of recursive equations and abstract away from the underlying Haskell semantics, in order to provide better error messages in case of instantaneous dependency – which are statically detected by all synchronous language compilers.

Elm is a functional language that targets non-critical reactive systems such as web services and GUIs [5]. The Elm’s debugger [21] thus focuses on interactions with the GUIs. Such a neat GUI integration is complementary to the program-based stochastic stimuli generation we propose.

Programmable debuggers. Several debuggers offer the possibility of defining debugging commands, taking advantage of existing programming environments and relying on an execution trace view as a sequence of runtime events.

The Prolog’s debugger Opium [7] and its derivatives (Coca [6] for C, Morphine [13] for Mercury) are based on a Prolog REPL. The debugger and the debuggee communicates via sockets, but some primitives perform as much event filtering as possible in the debuggee process to avoid context switches and inter-process communications. In Morphine, non-interactive commands (monitors) can be compiled and executed entirely in the debuggee process using dynamic linking.

MzTake [20] is programmable debugger for Java based on the FrTime [4], a functional reactive programming language based on Scheme. It communicates with the JVM via sockets (Java Debug Wire Protocol) and pays the price of 2 context switches per event.

Expositor [22] is a programmable debugger for C, based on GDB and UnixDB, a commercial replay debugger. The tool provides a Python API to program debugging commands. Typical Expositor program performs an initial query that computes a sub-trace of a complete execution trace. Then other queries can perform further filtering, merge several traces, or fold them. For efficiency, they use a mixture of lazy computations and replay debugging.

The moldable debugger [3] is a framework for smalltalk, that eases the development of domain-specific debuggers, and orchestrate their use. Such domain-specific debuggers are not really meant to be developed by end-users though.
8. CONCLUSION

We have presented RDBG, the first debugger for reactive programs that takes into account the feedback loop. We have demonstrated how to implement a full-featured debugger on top of a small kernel plugged onto 2 existing execution systems. This debugger is based on 4 simple and orthogonal ideas: (1) take advantage of existing languages (for the read-eval-print loop and the libraries); (2) provide users a pre-defined instrumentation and a simple API that allow them to implement their own debugging or monitoring commands; (3) provide compiler designers a plugin API; (4) use continuations to implement coroutine efficiently.

The advantages are multiple: a lightweight implementation (although, there remains technical work in the plugin implementation), which allow to program (or tune) efficient commands (no context switches nor inter process communications). This architecture offers a good separation of concerns, as debugging commands are implemented separately, and can serve for several language plugins. Besides, having the same infrastructure for testing and debugging is very handy: when the test exhibits a bug, everything is ready to launch a debug session.

RDBG has been designed for debugging reactive programs, that interact continuously with their environments. But RDBG can be used with non-reactive programs, which can be seen as degenerated kind of reactive programs that perform only step and that don’t interact with their environment (no input/output). More generally, part of the proposed infrastructure does not depend on the target language characteristics (Lustre or Lutin). It depends on the other hand on the host interpreter language (ocaml). However, any language equipped with a REPL and good libraries could be used as host language. In our implementation however, the fact that the host language, is also the language used to compile and interpret the targeted languages (Lustre and Lutin) facilitates the task.

The use of continuations to implement coroutine is not specific to any host or target language either. It might be easier with language where functions are first class citizens though.

RDBG can be used by any execution system that can be interfaced with ocaml. RDBG could therefore be a good starting point to provide to synchronous programs that do not have a debugger yet, in particular for languages which runtime system is already based on OCAML such as Lucid synchrone [23], ReactiveML [18], or Stimulus [17] – Stimulus is an industrial version of Lutin that is developed by the Argosim company.

9. REFERENCES