Work-in-Progress: mixing computation and interaction on FPGA

Loïc Sylvestre
loic.sylvestre@lip6.fr
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France

Emmanuel Chailloux
emmanuel.chailloux@lip6.fr
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France

Jocelyn Sérot
jocelyn.serot@uca.fr
Université Clermont Auvergne, CNRS,
Clermont Auvergne INP, Institut Pascal
F-63000 Clermont-Ferrand, France

ABSTRACT

This paper presents a programming language for the design and implementation of reactive embedded applications. The language is compiled to hardware descriptions for reconfiguring Field-Programmable Gate Arrays (FPGAs) using logic synthesis toolchains. It features synchronous semantics for fine-grained control on timing and parallelism in the applications. This enables interactions with physical I/Os to be safely composed with algorithms.

1 INTRODUCTION

When developing embedded applications, a classical problem is to combine *computations* (e.g., algorithms on data structures) and *interaction* with the physical environment (i.e., reading inputs and producing outputs in a reactive way). This is specially true when these applications are implemented on reconfigurable architectures such as FPGAs [3] and therefore encoded at the Register Transfer Level (RTL) using a global clock.

We are currently designing a programming language within which computations and interaction can be expressed in a unified way. This language, relying on the concept of *logical time* from synchronous languages [10], offers static guarantees of reactivity by compilation into synthesizable RTL code. Generated hardware designs can then be implemented on an FPGA while benefiting from a wide range of simulation and verification tools.

This paper gives an overview of the programming possibilities of this language on simple examples (Section 2). It then discusses related work (Section 3) and presents the current implementation of the language and the associated research perspectives (Section 4).

2 LANGUAGE OVERVIEW

The proposed language is a strict, statically-typed, functional-imperative programming language inspired by OCaml[1] and compiled into RTL. It supports tail-recursion (without requiring a stack) and data structures allocated in shared memory. The semantics of the language is synchronous: execution is formalized as a sequence of logical steps (or clock ticks). All language constructs react instantaneously (i.e., before the next tick) except tail-recursive function call (taking one tick) and memory accesses (which are asynchronous).

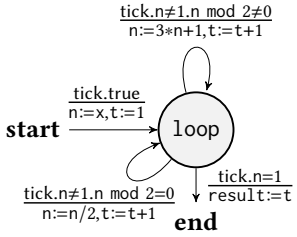
[1] https://ocaml.org

EMSOFT '23, September 17–22, 2023, Hamburg, Germany

ACM ISBN 979-8-4007-0291-4/23/09.
https://doi.org/10.1145/3607890.3608467

Each program is modeled as a clock-driven instantaneous function mapped to physical devices, as in synchronous dataflow programming [6]. This entry point can use auxiliary functions. Type-checking assigns distinct types to instantaneous functions (of type $\tau \Rightarrow \tau'$) and non-instantaneous functions (of type $\tau \rightarrow \tau'$), so that non-reactive programs can be statically detected and rejected.

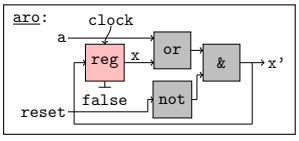
Expressing computations. On the left side of Figure 1 is a Finite State Machine (FSM) implementing a computation in RTL. The arrows of the diagram are transitions labelled with guarded actions $\frac{\texttt{tick.}condition}{action}$ meaning that on the next clock tick, if *condition* is true then *action* is executed. This FSM specifies an algorithm running for several logical steps until assigning a value to the variable `result` (which is undefined during the computation). On the right side of Figure 1 is a formulation of this algorithm, in our language, as a function `collatz` using a tail-recursive inner function `loop`. Synchronous semantics ensure that all tail-recursive function calls, like `loop(x,1)` and `loop(n/2,t+1)`, always pause until the next tick: the timing behavior of `loop` is equivalent to that of the FSM at left. This illustrates how tail-recursion is compiled into RTL.




```
let collatz(x) =
  let rec loop(n,t) =
    if n == 1 then t else
    if n mod 2 == 0 then
      loop(n/2,t+1)
    else loop(3*n+1,t+1)
  in loop(x,1)
val collatz : int → int
```

Figure 1: RTL implementation of a computation

Expressing interaction with physical I/Os. Reactive components, producing outputs synchronously to their inputs, are implementable at the RT level as synchronous Moore machines. On the left side of Figure 2 is for instance a block diagram specifying a synchronous circuit `aro` driven by an input `clock`. Circuit `aro` returns false until input `a` takes value `true`, and is reinitialized to false when `reset` takes value true. On the right side of Figure 2 is a formulation of this circuit, in our language, as an instantaneous function. This function is stateful: it uses a language construct (**`reg`** f **`last`** e) which represents the *next* input of a register initialized with expression e and updated with function f (where both f and e are instantaneous).




```
let aro(a,reset) =
  let step(x) =
    (x or a) & not reset in
  reg step last false
val aro : bool * bool ⇒ bool
```

Figure 2: RTL implementation of an interaction

Running computations in interactions. The language construct (**exec** e **default** e') computes asynchronously the expression e while providing a result at each tick. The result is either the value of the instantaneous expression e' (i.e., a default value) if the computation of e is still running, or the value of e if available. Once the computation of e terminates, it is restarted with current inputs. An extra output *rdy* takes value true at the very instants in which the computation of e terminates, and false otherwise. Figure 3 defines, for example, an instantaneous function `main(m,n,threshold,reset)` containing two **exec** constructs which progress independently. This function returns false until `collatz(m)` or `collatz(n)` produces a value higher than `threshold`, using the function `aro` defined Figure 2. This shows the inherent, fine-grained parallelism of the language as well as the ability to encode coarse-grained parallelism.

```
let main(m,n,threshold,reset) =
  let (t1,rdy1) = exec collatz(m) default 0 in
  let (t2,rdy2) = exec collatz(n) default 0 in
  let a = (rdy1 & (t1 > threshold)) or
          (rdy2 & (t2 > threshold)) in
  aro(a,reset)
val main : int * int * int * bool ⇒ bool
```

Figure 3: Mixing computation and interaction

Compilation into RTL. Well-typed programs are compiled down to RTL through semantics-preserving compilation passes:

(1) globalization of all functions (by *Lambda lifting*);
(2) inlining (i.e., replication) of all non-recursive functions;
(3) sharing of non-simultaneous tail-recursive function calls;
(4) translation to RT-level FSM with encoding of memory accesses as asynchronous bus transactions (using additional I/Os).

This approach allows the frequency of the clock tick to be automatically derived from the FPGA synthesis toolchain depending on the target's capabilities, without having to compute a Worst Case Execution Time (WCET) on the generated code [5, 9].

As an example of compilation, consider a sequential composition involving two calls to the function `collatz` defined Figure 1:

```
let z = collatz(y) in collatz(z)
```

The inner function `loop` is first globalized (1). Then, `collatz` is inlined (2), leading to:

```
let rec loop(n,t) = ··· (* same definition as on figure 1 *) in
let z = loop(y,1) in loop(z,1)
```

The two calls to `loop` are shared to limit resource usage (3):

```
type loop_instances = I0 | I1 (* the two instances of loop *)
(* Calls to be shared use an extra parameter to select the continuation *)
let rec loop(n,t,instance_id) =
  if n == 1 then (switch instance_id of
                  | case I0: let z = t in loop(z,1,I1)
                  | case I1: t)
  else if n mod 2 == 0 then loop(n/2,t+1,instance_id)
  else loop(3*n+1,t+1,instance_id)
in loop(y,1,I0)
```

Translation into RTL (4) is then similar to those illustrated Figure 1.

3 RELATED WORK

Synchronous languages are a proven technology for the design of reactive embedded systems [2]. Esterel [4, 7] and Lustre [9], in particular, have been used to generate RTL code mapping logical time to the clock of an FPGA. Our work is based on this approach.

Lustre modelizes discret systems using instantaneous computation steps. We generalize this programming style to express non-instantaneous computations. Extensions of Lustre like Scade use type-based analyses [6] similar to the static analysis we propose. These also offer control structures like state machines and futures [5], what could be profitably implemented in our work.

Chisel [1] is a framework for RTL description, simulation and verification in the Scala programming language. It offers software abstractions, such as Scala classes, to improve RTL code reuse and portability. In contrast, we propose a cycle-accurate programming language compiled to RTL for the design of embedded systems.

4 IMPLEMENTATION AND PERSPECTIVES

An experimental tool, based on the ideas of this paper, is available online (https://github.com/lsylvestre/EMSOFT23). It comprises:

- a type checker to ensure type safety and reactivity;
- an interpreter using the formal semantics of the language;
- a VHDL code generator for logic synthesis and simulation.

We are currently working on the integration of this compiler onto an FPGA-toolchain using O2B [11], which is a softcore implementation of the OCaml Virtual Machine. Reactive programs are directly mapped to physical I/Os. Non-reactive functions can access shared memory through a bus. Transactions on this bus and memory management remain sources of unpredictability to be analyzed [8].

Future evaluation on large applications mixing computation and interaction will show if the proposed approach is applicable to the design of mixed critical embedded systems on FPGA.

REFERENCES

[1] J Bachrach et al. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*, 1216–1225.
[2] A Benveniste et al. 2003. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91, 1, 64–83.
[3] C Bernardeschi, L Cassano, and A Domenici. 2015. SRAM-based FPGA systems for safety-critical applications: A survey on design standards and proposed methodologies. *Journal of Computer Science and Technology (JCST)*, 30, 2, 373.
[4] G Berry. 1992. A hardware implementation of pure Esterel. *Sadhana*, 17, 95–130.
[5] A Cohen, L Gérard, and M Pouzet. 2012. Programming parallelism with futures in Lustre. In *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT '12)*, 197–206.
[6] J-L Colaço, A Girault, G Hamon, and M Pouzet. 2004. Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM international Conference on Embedded Software (EMSOFT '04)*, 230–239.
[7] J Hammarberg and S Nadjm-Tehrani. 2003. Development of safety-critical reconfigurable hardware with Esterel. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 80, 219–234.
[8] F Restuccia et al. 2019. Is your bus arbiter really fair? restoring fairness in axi interconnects for FPGA SoCs. *ACM Transactions on Embedded Computing Systems (TECS)*, 18, 5s, 1–22.
[9] F Rocheteau and N Halbwachs. 1992. Implementing reactive programs on circuits a hardware implementation of Lustre. In *Real-Time: Theory in Practice: REX Workshop Mook 1991*. Springer, 195–208.
[10] A Schulz-Rosengarten, S Smyth, R von Hanxleden, and M Mendler. 2018. On reconciling concurrency, sequentiality and determinacy for reactive systems—A sequentially constructive circuit semantics for Esterel. In *18th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 95–104.
[11] L Sylvestre, E Chailloux, and J Sérot. 2023. Accelerating OCaml programs on FPGA. *International Journal of Parallel Programming (IJPP)*, 51, 2-3, 186–207.