Concurrent Programming of Microcontrollers, a Virtual Machine Approach

Steven Varoumas, Benoît Vaugon, Emmanuel Chailloux

Concurrent Programming of Microcontrollers, a Virtual Machine Approach

Steven Varoumas[1,2], Benoît Vaugon[3] and Emmanuel Chailloux[2]

[1]CÉDRIC – Conservatoire national des arts et métiers, Paris, F-75141, France

[2]Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606, 4 place Jussieu 75005 Paris, France.
steven.varoumas@lip6.fr, emmanuel.chailloux@lip6.fr

[3]U2IS, ENSTA ParisTech, Palaiseau, France
benoit.vaugon@ensta.fr

Abstract

Microcontrollers are low-cost and energy efficient programmable integrated circuits, they are used in a lot of common electronic devices but are quite difficult to program because of very limited resources. Being particularly used for embedded system, they interact a lot with their environment, and should react quickly to external stimuli. In this paper, we study different models of concurrency for programming microcontrollers using a virtual machine approach for safety as well as a higher-level model of programming. We then propose OCaLustre, the prototype of a synchronous extension to OCaml suitable for concurrent programming on microcontrollers.

1 Introduction

Microcontrollers are small integrated circuits that can be considered as simple, albeit complete, computers: they contain a processing core, multiple memory units (typically, nonvolatile memory for program code and volatile memory for data) as well as a set of input/output pins which allow interactions with the surrounding environment of the chip by conducting electric current. Being very small, affordable and energy efficient, microcontrollers are ubiquitous in embedded systems: they can be found implanted in the electronics of common-life objects (such as domestic appliances or toys), as well as in bigger, critical, machines (manufacturing robots, car engines, aircraft systems, ...), on which they perform tasks of various nature and complexity. Those efficiency advantages come with some drawbacks: microcontrollers typically offer limited processing power and low memory resources, constraining programmers of microcontrollers to use low level programming models in order to keep permanent control of the available hardware resources, memory consumption in particular.

Due to simplicity and performance concerns, programs running on microcontrollers are commonly written in Basic, assembly or subsets of the C language. These programming languages

often fail to provide the same level of hardware abstraction, safety, and expressiveness as higher-level programming languages such as Python, Lisp, Java, or OCaml. From this observation, a few virtual machines capable of interpreting the bytecode of such languages have been successfully ported to microcontrollers. These solutions free developers from a lot of hardware considerations, and allow the development of less error-prone and more complex software, providing levels of hardware abstraction, increased safety and a more modern programming style overall.

Typically, embedded systems in which microcontrollers are operating are regularly interacting with the outside world, often reacting to signals sent by various peripherals controlled or not by humans (buttons, sensors, other controllers, etc.). In a lot of cases, all of those different stimuli must be acknowledged and treated as they appear and in any particular order, leading the programs running on microcontrollers to be inherently concurrent. Unfortunately, none of traditional ways of programming microcontrollers (namely imperative low level languages), nor the languages provided by virtual machine approaches are particularly suited for the handling of concurrent tasks for such systems. We thus intend to expand the ways microcontrollers are developed to better comply with the nature of embedded systems, and do so while keeping the safety and expressivity of high-level programming languages as well as a small resources consumption thanks to bytecode factorization and automatic memory management.

In this paper, we study the different ways to improve the programming of microcontrollers with concurrent programming, using a particular virtual machine capable of handling bytecode of the OCaml programming language. We especially focus on providing a high-level model of concurrency adapted to embedded systems and with a low memory consumption in order to comply with the limited resources of microcontrollers. Our different efforts, associated with the development of the OCaPIC virtual machine, lead us to develop OCaLustre, a prototype of a synchronous data-flow extension to OCaml.

2 A virtual machine approach for more powerful programming languages

Resources on microcontrollers, especially memory space, can be very low. For example, the PIC 18 series of microcontrollers commercialized by the *Microchip* company can have at most only 4 kibi-bytes (KiB) of RAM and 128 KiB of program memory. These constraints usually drive the style of programming for such devices to be very low level, with a manual handling of memory usage. For this reason, most microcontrollers are programmed with a subset of C or assembly languages, and programmers need to be knowledgeable about the precise architecture of the chip they use. Not only being quite tedious, not portable, and lacking the richness of constructions of higher-level languages, these approaches also lack important programming safeguards such as static type checking at compile-time or automatic memory management at runtime, leading to the apparition of unforeseen bugs and issues.

In order to free programmers from dealing with tedious tasks relating to the hardware, and to help them develop safer programs, some ports of virtual machines capable of running the bytecode of higher-level programming languages have been completed. These virtual machines, directly implemented in the lower level languages traditionally used for microcontrollers' programming, allow a more portable code and a safer programming model while staying fast and efficient.

Among these various virtual machine approaches, we can mention the Darjeeling Virtual Machine (DVM) [1], a port of the Java virtual machine on Atmel and ARM microcontrollers capable of running a subset of this language. Similarly, the PICBIT [4] and PICOBIT [10] systems allow to run Scheme programs on PIC microcontrollers with virtual stack machines.

Quite paradoxically, these virtual machines approaches can lead to a smaller resulting code

(that includes the runtime library and virtual machine) than the corresponding native code because of the more powerful and more complex instruction set of the virtual machines and thanks to bytecode compression and cleaning tools.

OCaPIC: running OCaml bytecode on PIC microcontrollers

The OCaPIC project [11] is a virtual machine approach directed towards running bytecode of the OCaml programming language on the very limited hardware of the PIC 18 microcontrollers. This port of the ZINC Abstract Machine (ZAM) [6] (the original virtual machine of OCaml), written in PIC assembly, allows programmers to use the various advantages of this language and of its runtime on microcontrollers with very scarce resources.

OCaml is a high-level programming language belonging to the ML family of programming languages. Descending from Caml and Caml-light, it was created and maintained at INRIA [1] since 1996. Being multi-paradigm, it implements functional, imperative, modular and object-oriented traits and thus offers a rich expressiveness for writing programs of various nature for embedded systems. Furthermore, OCaml provides a strong static type-checking at compile time, with type inference, which insures the absence of dynamic type error and memory corruption, and thus decreases the amount of possible bugs inside critical applications. Moreover, OCaml comes with a garbage collector (GC) which makes possible an automatic handling of memory resources, and frees programmers from such considerations while providing re-usability of memory. OCaPIC implements two different algorithms of GC: stop-and-copy (by default) and mark-and-compact.

In addition to the port of most of the standard library of OCaml, OCaPIC offers a set of primitives adapted to the handling of the input/output pins: for example, calling the function `set_bit` makes the microcontroller send an electric current on the pin passed in argument and calling `test_bit` reads and returns the state of a given pin.

The following code is an example of a program written with OCaPIC which makes a LED connected to the pin named `RB0` blink every second.

```
open Pic;; (* Module containing write_reg, set_bit, RB0, ... *)
write_reg TRISB 0x00; (* Configure the port B to be an output *)
while true do
    set_bit RB0;
    Sys.sleep 1000;
    clear_bit RB0;
    Sys.sleep 1000;
done
```

After compiling an OCaml program with the standard compiler (`ocamlc`), the resulting bytecode is then cleaned by the provided OCamlClean tool which removes residual dead code. This cleaning operation also has an interesting impact at runtime: the heap is cleaned from some closures and unused global data, typically coming from unused parts of libraries. The resulting bytecode is then compressed and linked with its interpreter into an hexadecimal file that can finally be flashed on a PIC microcontroller. This workflow is depicted in the figure 1.

Finally, OCaPIC comes with two different simulators: the first one interprets the bytecode and makes possible an easy debugging process using usual OCaml tools for correcting errors, such as `ocamldebug`. The second simulator interprets the hexadecimal file produced by OCaPIC and emulates the physical capacities of the microcontroller: it allows checking the native runtime and the different kind of arithmetic and memory overflows. These simulators give a graphical representation of the state of each pin of the microcontroller, in order to check the coherence of

[1] Institut National de Recherche en Informatique et en Automatique

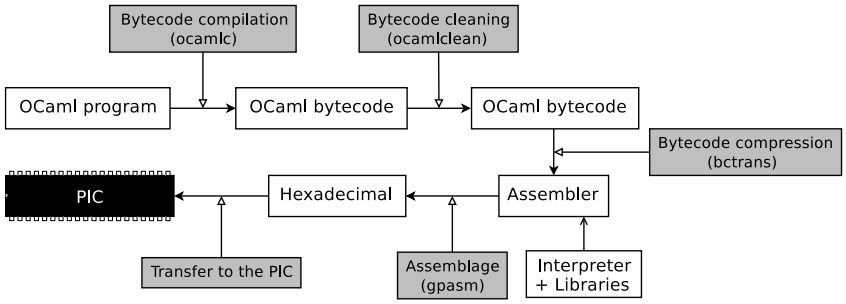

Figure 1: The OCaPIC compilation chain, from OCaml to PIC

the input and output of the programs. They are also able to simulate simple electronic boards connected to the chip (LCD displays, buttons, ...) for testing purposes.

3 Models of concurrent programming

We engage in improving the virtual machine approach in order to better comply with the nature of embedded programs. Because those systems are in constant interaction with the outside world, for example reacting to button presses, impulses from sensors or signals emitted by other computational devices, we focus on finding a model of concurrent programming adapted to the scarce resources of microcontrollers that could run in the OCaPIC virtual machine.

Analysis of Models of Concurrent Programming

To achieve this goal, we experimented various models of concurrency and analyze their resources consumption and expressiveness.

Firstly, preemptive threading approaches seems to be an ill-adapted model because our microcontrollers do not run any operating system and thus do not provide any underlying scheduling features capable of switching context between tasks depending on priority. Threads directly scheduled by the VM could be conceivable, however they would be quite demanding on memory resources. Moreover, this model of concurrency is not easily predictable and thus any kind of static analysis capable of checking that programs do not go wrong is very difficult to achieve.

Cooperative threading is a far better candidate: letting each process explicitly hand control to other threads does not need the presence of an operating system. We have been successful when porting the core parts of the LWT cooperative thread library [12] into OCaPIC. Reacting to environment stimuli can be done by frequently polling the value on each entry pin of the microcontroller in a separate thread but handling the control points of programs is quite tedious and a single process can block all the system if it does not yield control to the others.

We also managed to port the React functional reactive programming module[2] to OCaPIC, providing a model of concurrency using signals and events appearing and changing through time. While appearing lightweight, this library makes a heavy use of memory allocation, and despite our efforts, we were unable to keep memory use limited to the hardware restrictions.

Lastly, a port of the ReactiveML [7] synchronous language has been performed, but its heavy use of OCaml functors creates a need for too much memory and prevents it for being a viable solution for the concurrent programming of microcontrollers.

[2]See http://erratique.ch/software/react

Nonetheless, the synchronous approach appears to be well-suited for our goal and we decided to direct our work from the control-flow model of ReactiveML to a data-flow synchronous concurrent model.

4 OCaLustre: a synchronous extension to OCaml

All of those experiments have exhibited some major drawbacks for the use of the aforementioned concurrent programming models in real-life applications such as efficiency consideration as well as expressiveness concerns. We thus propose OCaLustre, a synchronous extension to OCaml based on the Lustre [3] dataflow synchronous programming language.

4.1 Syntax and Semantics of OCaLustre

An OCaLustre program is an OCaml program augmented with a construct stemming from Lustre: the node. Nodes are synchronous functions operating on data flows which are considered as being executed instantaneously, following the synchronous hypothesis. The body of each node is equivalent to a system of equations that is solved between two *"ticks"* of the reactive system and these equations, being solved all at once during the same instant, can represent concurrent tasks. We believe that this model is well-adapted to microcontrollers' programming because, at any moment, each of the input/output pin of a chip is either holding an electrical current or not: the absence or presence of current can thus be represented as a boolean data flow and treated by such nodes in order to react "instantaneously" to environment stimuli. The complex algorithmic and computational parts of programs are still handled by traditional OCaml functions that are called inside nodes in order to keep the advantages of the expressiveness of high level constructs.

The syntax of OCaLustre nodes is very close to Lustre, and users familiar with the Lustre style of programming will not really be surprised when using OCaLustre: for example, the code of figure 2 declares a node called `count_pairs` that computes the series of all natural numbers, as well as all the even numbers. The complete syntax for OCaLustre nodes is given on figure 3.

```
let%node count_pairs () (n, m) =
    m := n * 2;
    n := 0 --> pre n + 1
```

Figure 2: An OCaLustre node

All of the OCaml boolean and arithmetic operators can be used inside OCaLustre declarations, as well as the initialization (`->`) and memory (`pre`) operators of Lustre. Note however that the operator `->`, being already reserved by OCaml, has been replaced by `-->` to describe the initialization of data flows. The `when` operator is not usable at this moment, because our OCaLustre prototype doesn't handle nodes running on different clocks (they all run on the default clock). Notice also that type annotations are not requested in OCaLustre, because it is compiled into pure OCaml code that offers a type inference mechanism. In this regard, OCaLustre nodes can handle polymorphic flows, as it is the case in the following code that declares a node testing if the value of a flow `f` has changed between two instants.

```
let%node change (f) (changed) = changed := false --> f <> pre f
```

Adding a new behavior to an existing OCaLustre program is very straightforward: one just have to write new equations providing the new behavior into the involved node(s). Figure 4 presents such a modification: a program waiting for two signals `a` and `b` and returning `o` only

$$
\begin{aligned}
< node > &::= \texttt{let\%node}\ node\_id < flows\_sig > < flows\_sig > \ \texttt{=}\ < decl\_seq > \\
< flows\_sig > &::= \texttt{(}id[\texttt{,}id]*\texttt{)} \mid \texttt{()} \\
< decl\_seq > &::= < declaration > \mid < decl\_seq > \texttt{;} < declaration > \\
< declaration > &::= < flow > \ \texttt{:=}\ < expression > \\
< flow > &::= id \mid \texttt{(}id\texttt{,}\ id\texttt{)} \\
< infix\_op > &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{+.} \mid \texttt{-.} \mid \texttt{*.} \mid \texttt{/.} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{\&\&} \mid \texttt{||} \\
< prefix\_op > &::= \texttt{not} \\
< constant > &::= int \mid bool \mid float \mid \texttt{()} \\
< parameters > &::= \texttt{(}< parameter > [\texttt{,}< parameter >]*\texttt{)} \mid \texttt{()} \\
< parameter > &::= id \mid < constant > \\
< expression > &::= < constant > \mid id \\
&\quad \mid \texttt{if}\ < expression > \ \texttt{then}\ < expression > \ \texttt{else}\ < expression > \\
&\quad \mid < expression > < infix\_op > < expression > \\
&\quad \mid < expression > \texttt{-->} < expression > \\
&\quad \mid < prefix\_op > < expression > \mid \texttt{pre}\ < expression > \\
&\quad \mid \texttt{call}\ ocaml\_function\_id < parameters > \\
&\quad \mid node\_id < parameters > \mid \texttt{(}< expression > \texttt{,} < expression > \texttt{)}
\end{aligned}
$$

Figure 3: The syntax of OCaLustre

when `r` is not present is easily expanded into a program waiting for the presence of a third signal `c`. This process doesn't lead to any explosion of memory usage when the resulting program is executed, due to the lightweight model of compilation of Lustre.

```
let%node edge (x) (y) =
   y = false --> x && not pre x

let%node abro (a,b,r)  (o) =
   o := edge (seenA && seenB);
   seenA := false -->
      not r && (a || pre seenA);
   seenB := false -->
      not r && (b || pre seenB)
```

```
let%node abcro (a, b, c, r) (o) =
   o := edge (seenA && seenB &&
              seenC);
   seenA := false -->
      not r && (a || pre seenA);
   seenB := false -->
      not r && (a || pre seenB);
   seenC := false -->
      not r && (c || pre seenC)
```

Figure 4: Adding a new behavior to an OCaLustre node

4.2 Compilation of OCaLustre

An OCaLustre program is compiled following the Lustre model of "simple-loop" compilation: each node is converted into a sequential function, and the main node of the program is run inside a single endless loop. Following this method, an OCaLustre program is compiled into a pure OCaml program that can then be handled by OCaPIC as any other OCaml program (see figure 5). Note that, because these two steps are entirely independent, the OCaml code produced after compilation of OCaLustre can also be used with any other OCaml interpreter or compiler: this makes possible to use our extension with any kind of hardware target supported by OCaml.

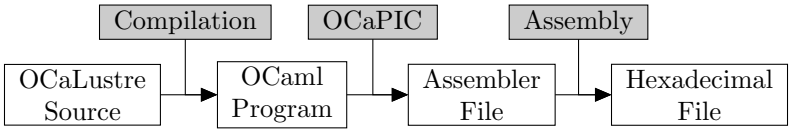

Figure 5: From OCaLustre to executable through pure sequential OCaml

During compilation, each node is replaced by an OCaml function by converting all of the declarations contained inside its body into a sequence of assignations. In OCaLustre, the equation order does not matter (we may use a flow before declaring it) but in OCaml this order matters and a reordering of each declaration is completed at compile-time. This reordering corresponds to a rescheduling of each task of our concurrent program. This process may fail when two flows depend on each other at the same instant: this kind of behavior is invalid since it denotes a causality loop inside the program and an error is then raised by the compiler.

Each usage of the memory operator "`pre`" is converted into an OCaml reference holding an option type. At the first instant, all references hold the value `None`, and at instant $i+1$, they contain the value of its flow at instant i.

The `-->` initialization operator is converted into a simple conditional operator checking with a simple boolean value `init` if the function is executed at the first instant or not.

In order to preserve the execution context of a node, a closure is returned by each instantiation function corresponding to a synchronous node. This closure is bound to all of the registers of the node. All of these compiling rules are depicted in the example of figure 6 which illustrates code generation for the node `count_pair` of figure 2.

```
let count_pairs () =
   let init = ref true in
   let pre_n = ref None in
   let count_pairs_step () =
      let n = if !init then 0 else Option.get (!pre_n) + 1 in
      let m = n * 2 in
      init := false; pre_n := Some n; (n, m) in
   count_pairs_step
```

Figure 6: Compilation of an OCaLustre node

4.3 Benchmark analysis

The compilation model of OCaLustre is quite efficient and makes the resulting compiled programs very small. In order to compare the efficiency of OCaLustre with the various models of concurrency that we had firstly considered, we developed a simple application in which two different counters are concurrently displayed on an LCD screen. Four versions of this application have been developed with OCaLustre as well as with three other models: cooperative threading with LWT, functional reactive programming with React and a simple sequential program written in pure OCaml. The size of the programs generated by OCaPIC after a cleaning pass by OCamlClean (containing the runtime library, the virtual machine as well as the compressed bytecode) allows us to assert that OCaLustre is a very lightweight solution, using about 1.5 times less space than LWT and being more than three times smaller than the React solution. In fact, the OCaLustre application is very close, in size, to the compiled sequential code. We executed the programs on our microcontroller (a PIC 18F4620) and analyzed the dynamic memory allocation of each tool when the program is loaded. We found again that OCaLustre is very low demanding

in resources with React and LWT allocating respectfully 1668 and 1136 bytes while OCaLustre only using 272 bytes of dynamic memory space:

Tool	React	LWT	OCaLustre	Sequential Code
Size of the program	23.8 KiB	11.7 KiB	**7.8 KiB**	6.8 KiB
Initial dynamic allocation	1668 B	1136 B	**272 B**	150 B

Using our OCaLustre prototype, we developed a program for a device capable of tempering chocolate. This device receives inputs from a temperature sensor and from two buttons (labeled "+" and "-") used to set a desired temperature for the chocolate. Its outputs are an LCD displaying the desired temperature, as well as the current temperature of the chocolate, and a set of resistances capable of heating the preparation. The following OCaLustre program controls the tempering machine by computing a value (`prop`) that represents the amount of time at which the resistances need to be on, depending on the difference between the desired temperature (`wtemp`) and the actual temperature (`ctemp`). The `main` node of the program receives at each instant the value of the buttons as well as the current temperature, computes the value of the desired temperature, the state of the device (on or off) and decides if the heating resistances must be active based on all of these informations:

```
(* Temperature in celsius is (1033-ctemp)/11.67 *)
let%node update_prop (wtemp, ctemp) (prop) =
    new_prop := 0 --> (min (100, max (0, pre new_prop + offset)));
    delta    := min (10, max (-10, ctemp - wtemp));
    delta2   := if delta < 0 then -delta * delta else delta * delta;
    offset   := min (10, delta2);
    prop     := new_prop / 10

let%node timer (number) (alarm) =
    time  := 1 --> if pre time = 100 then 1 else pre time + 1;
    alarm := time < number * 10

let%node heat (w, c) (h) =
    prop := update_prop (w, c);
    h    := timer (prop)

let%node change_wtemp (state) (w) =
    w := 654 --> if state = 1 then pre w - 1 else
                 if state = 2 then pre w + 1 else pre w

let%node thermo_on (state) (on) =
    on := true --> if state = 3 then not (pre on) else pre on

let%node main (plus, minus, ctemp) (wtemp, on, heat) =
    state := call (buttons_state plus minus);
    on    := thermo_on (state);
    wtemp := if on then change_wtemp (state) else 0;
    heat  := if on then heat (wtemp, ctemp) else false;
```

An application displaying the same behavior was also developed in classic OCaml and used in OCaPIC before the development of OCaLustre, so we are able to compare the size of our OCaLustre chocolate tempering program versus the size of the OCaml one, as well as display the significant size difference between the generated OCaml bytecode and the resulting compressed OCaPIC program:

Language	OCaml		OCaLustre	
Type	Bytecode File	PIC Program	Bytecode File	PIC Program
Size	268 KiB	27 KiB	258 KiB	**27 KiB**

We conclude from these results that, after cleaning and compression, the PIC program is a lot smaller than the original bytecode file. Judging by various experiments, this decrease by a factor of about 10 is not uncommon. We show once again that the OCaLustre mode of compilation keeps programs very lightweight and thus allow to run real world programs on devices with small resources.

5 Conclusion

Our efforts dedicated at proposing a higher level of programming for microcontrollers seem promising. In particular, OCaLustre, a synchronous extension to the multiparadigm OCaml programming language, makes possible an easier development of concurrent programs aimed at embedded systems. This extension, based on the data flow language Lustre is quite similar to the Lucid Synchrone programming language [2] but our prototype offers a simpler model. It is translated into pure OCaml and compiled into bytecode that can be lightened by OCamlClean. This model makes the program written in OCaLustre smaller than the ones developed in the richer Lucid language which unfortunately (and contrary to OCaLustre[3]) is not distributed with an open source license, which was necessary for our early experiments with models of concurrency.

Real-time considerations for programs developed with OCaLustre are usual: Worst Case Execution Time (WCET) of the synchronous parts of OCaLustre programs can be computed using classical tools for synchronous programming. For the algorithmic part written in OCaml, we need a guarantee that there is enough memory and to consider dynamic memory management (garbage collector or GC) execution time for WCET analysis. These two notions are nested but depend on the programming style of the algorithmic part and need to deal with dynamic memory allocations (stack and heap) for recursive functions and dynamic data structures. One possibility will be to use a real-time GC [5] but they have important costs (time and memory) and not really adequate for the low resources of PIC micro-controllers. For that, we need to help the GC by giving indications on liveness of objects. Some on-going works can use static region analysis for a mini-ML. In this case freeing a region can be immediate or can allow to manually trigger the garbage collector. The virtual machine approach is not detrimental to the real-time aspects, actually it also makes possible the factorization of the WCET by measuring it directly on the bytecode and then extrapolate it for different hardware.

Moreover, the use of a virtual machine facilitates the debugging and tracing of programs: without modifying program sources the interpreter can be instrumented in order to offer useful informations, allowing a non-intrusive structural code coverage like Zamcov project [13] for the Ocaml Virtual Machine (ZAM). The association of functional languages with critical embedded systems was not previously unseen and have already been used (with less hardware constraints) for the development of various tools, in particular of code generators (like KCG, the code generator of the SCADE SUITE™ [8]).

Using this virtual machine approach, we are able to produce lightweight and portable code which offers higher-level guarantees thanks to the use of the OCaml programming language. The different tools of code analysis are also factorized by the use of bytecode and can be adapted for many different microcontrollers, offering more safety for embedded system applications. Finally, our synchronous extension of OCaml offers a deterministic model of concurrence adapted to the nature of embedded system and the analysis of the safety of programs and its association with an OCaml virtual machine makes possible the development of richer and safer applications.

We aim for future works at improving the OCaLustre language and developing more serious applications (for example in robotics or home automation) while trying to offer formal guaran-

[3] `https://github.com/stevenvar/OCaLustre/`

tees for the execution of programs inside devices fitted with minimal memory and computing resources. As for OCaPIC, a couple of similar projects aimed at porting the virtual machine of OCaml for Atmel AVR and STM32 microcontrollers (respectively used in Arduino and Nucleo platforms) have been instantiated.

References

[1] Brouwers, N., Corke, P., and Langendoen, K. Darjeeling, a Java Compatible Virtual Machine for Wireless Sensor Networks. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion* (2008).

[2] Caspi, P., Hamon, G., and Pouzet, M. *Real-Time Systems: Models and verification — Theory and tools*. ISTE, 2007, ch. Synchronous Functional Programming with Lucid Synchrone.

[3] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. Lustre: A declarative language for real-time programming. In *In 14th Symposium on Principles of Programming Languages (POPL'87). ACM* (New York, NY, USA, 1987), POPL '87, ACM, pp. 178–188.

[4] Feeley, M., and Dubé, D. Picbit: a Scheme system for the PIC microcontroller. In *Scheme and Functional Programming Workshop (SFPW'03)* (Nov. 2003), pp. 7–15.

[5] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Jan. 2012.

[6] Leroy, X. The ZINC experiment : an economical implementation of the ML language. Tech. Rep. RT-0117, INRIA, Feb. 1990.

[7] Mandel, L., and Pouzet, M. ReactiveML, a reactive extension to ML. In *Proceedings of 7th International conference on Principles and Practice of Declarative Programming (PPDP 2005)* (Lisbon, Portugal, July 2005).

[8] Pagano, B., Andrieu, O., Moniot, T., Canou, B., Chailloux, E., Wang, P., Manoury, P., and Colaço, J.-L. Experience Report: Using Objective Caml to develop safety-critical embedded tool in a certification framework. In *International Conference of Functional Programming ICFP 09* (2009), pp. 215–220.

[9] St-Amour, V., and Feeley, M. Picobit: A Compact Scheme System for Microcontrollers. In *International Symposium on Implementation and Application of Functional Languages (IFL'09)* (Sept. 2009), pp. 1–11.

[10] Vaugon, B., Wang, P., and Chailloux, E. Programming Microcontrollers in Ocaml: the OCaPIC Project. In *International Symposium on Practical Aspects of Declarative Languages (PADL 2015)* (June 2015), no. 9131 in Lecture Notes in Computer Science, Springer Verlag, pp. 132–148.

[11] Vouillon, J. Lwt: A cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML* (2008), ML '08, ACM, pp. 3–12.

[12] Wang, P., Jonquet, A., and Chailloux, E. Non-Intrusive Structural Coverage for Objective Caml. In *5th Workshop on Bytecode Semantics, Verification, Analysis and Transformation* (2011), vol. 264 4 Electronic Notes in Theoretical Computer Science, Elsevier, pp. 59–73.