Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems *

Nicolas Berthier
UJF/Verimag †
nicolas.berthier@imag.fr

Florence Maraninchi
Grenoble INP/Verimag
florence.maraninchi@imag.fr

Laurent Mounier
UJF/Verimag
laurent.mounier@imag.fr

Abstract

In embedded systems, controlling a shared resource like the bus, or improving a property like power consumption, may be hard to achieve when programming device drivers individually. There is a need for *global resource control*, taking decisions based on a centralized view of the devices' states. In this paper, we study power consumption in sensor networks, where the nodes are small embedded systems powered by batteries. We concentrate on the hardware/software architecture of a node, where significant gains can be achieved by controlling the consumption modes of the various devices globally. The architecture we propose involves a simple adaptation of the application level, to communicate with the hardware via a *control layer*. The control layer itself is built from a set of simple automata: the drivers of the devices, whose states correspond to power consumption modes, and a controller that enforces global properties. All these automata are programmed using a synchronous language, whose compiler performs static scheduling and produces a single piece of C code. We explain the approach in details, demonstrate its use with either Contiki or a traditional multithreading operating system, and report on our experiments.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.1 [*Programming Techniques*]; D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.2.11 [*Software Engineering*]: Software Architectures; D.4.7 [*Operating Systems*]: Organization and Design

General Terms Algorithms

Keywords Synchronous Paradigm, Automated Control, Power-Aware Implementation, Wireless Sensor Networks

* This work has been partially supported by the French ANR project ARESA2 (ANR-09-VERS-017).

† Verimag is an academic research laboratory affiliated with: the University Joseph Fourier Grenoble (UJF), the National Center for Scientific Research (CNRS) and Grenoble Polytechnic Institute (Grenoble INP).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.
Copyright © 2011 ACM 978-1-4503-0555-6/11/04. . . $10.00

1. Introduction

1.1 Resource Control in Embedded Systems

In embedded systems, controlling a shared resource, or improving a global property, may be crucial in some application domains. As an example, consider power consumption in the node of a wireless sensor network (WSN). A node is a small embedded system powered by a battery that cannot be recharged. Optimizing power consumption has a direct effect on the lifetime of the system. This is a typical *cross-layer* problem, because all the elements of a network have some impact: the hardware devices, the protocols, and the application. To start with, the choice of the hardware devices, like the radio component or the micro-controller (MCU), is very important. Choosing devices that can be put in some low consuming mode when nothing happens may offer significant gains in sensor networks, where traffic is quite low.

Once the hardware devices have been selected, programming an embedded node in such a way that the low-consumption modes of the various devices be well exploited is not easy. The software has a huge impact on the consumption states of the various devices (e.g., the driver of the radio puts it in the low consumption mode), but this low-level software is usually designed in a local, per device, way. Information on the consumption states of the various devices is then scattered among several pieces of code (device drivers, protocols, application, or operating system), and the decisions are necessarily taken in a local, decentralized manner.

1.2 Need for Global Control

Let us look at an example that illustrates common problems with decentralized approaches for controlling the power modes of the hardware devices.

Consider a simple sensor network application involving two concurrent tasks. The first one periodically senses the environment using a passive sensor connected to an analog-to-digital converter (ADC), and sometimes stores this information by using a flash memory module. When the collected data satisfy a given property, an alarm is sent to a special node of the network (the *sink*). The second task manages the network by listening to a channel, with the help of a radio transceiver device. This part of the software is responsible for routing the packets received to the desired nodes, and sending new alarms upon request from the first task.

Numerous existing sensor network hardware platforms ("mote") provide several devices connected to the micro-controller unit (MCU) using a limited number of buses. Therefore, the two almost independent tasks are in practice constrained by *shared resources* like buses. Concurrency problems must then be avoided by enforcing *global* properties such as mutual exclusion of the accesses to shared resources.

Another example deals with the reduction of instantaneous power consumption in such systems. For instance, one would like

to ensure that the radio and the flash memory devices are not simultaneously in an energy-greedy operating mode. If the control of the radio (resp. the memory) is implemented in the driver of the radio (resp. the memory), the decisions on the total power consumption cannot be done, because there is no place in the software where the global information is available.

With *global control*, the idea is that all information on the power modes of the devices should be gathered and exploited by a power-consumption policy implemented as a *centralized controller*.

1.3 Problem Formulation and Proposal

The problem we consider in this paper is the following: given a hardware architecture made of several devices whose consumption states are known, plus some existing application software (*e.g.,* the protocol stack in the case of sensor networks nodes), how to replace the low-level software (typically the set of drivers) by a *control layer* that implements a global power consumption policy? This should be done in such a way that only very small changes are required in the existing application software.

In our proposal, the application software can be implemented as a set of threads on top of a scheduler, or with event-driven programming as in Contiki [10].

To build the control layer we use *synchronous programming*, which has been studied a lot in the embedded system community, especially for hard real-time safety-critical systems, like nuclear plant controllers or automatic flight control. The family of *synchronous languages* [4] offers complete solutions, from pure static scheduling to some operating system support. The control layer is designed as a parallel program in a synchronous language, and then statically scheduled by the compiler to produce sequential C code.

The design of the control layer is inspired by *controller synthesis* techniques [21], although we do no use a controller synthesis tool, for efficiency reasons. The approach is similar to several proposals that have been made in the family of synchronous languages and tools (see, for instance, [7], [16] or [3]).

To summarize, the approach is as follows: (i) each device driver is described as a simple Mealy automaton M_i, whose transitions are labeled by Boolean formulas made of inputs from both the hardware and the software, and by outputs representing low-level C code; (ii) we specify some global properties, like: "*P: the devices A and B should not be in their highest consuming modes at the same time*"; (iii) the automata of the drivers are made *controllable*, meaning that the absence of an additional input may prevent the automaton from changing states; this yields the family of automata M_i''s. (iv) we build an automaton C, to control the M_i''s in such a way that global properties like P are ensured (this is typically where a controller synthesis tool would be used); (v) C and the M_i''s are programmed in some synchronous language; (vi) the control layer is obtained by *compiling* the parallel composition of C and the M_i''s into a single piece of sequential C code.

This approach has some consequences on programming models. Indeed, the control layer may refuse to execute a command that changes the state of a device, when this would result in a global state forbidden by some property like P. In the implementation of the control layer, one may choose to cancel the request, or to delay it until it becomes acceptable. In either cases, the consequences on programming models are similar to what has to be done for error handling.

The whole approach allows to reuse a wide range of previously written software and operating systems, by replacing some of their device drivers code with requests to the control layer. Porting an operating system originally designed for sensor networks to our new hardware/software architecture requires the replacement of the software that drives the physical devices such as the radio transceiver and the flash memory modules, or that manages a bus. Higher level parts like the network stack and the file systems remain in the original operating system.

1.4 Contributions and Structure of the Paper

This paper makes three contributions to global resource control in embedded systems: (i) a software architecture based on a *control layer*, between the hardware and the high level software; (ii) a method for obtaining the control layer automatically from a formal description of the device drivers, plus a description of the global properties that should be ensured; (iii) a working implementation of these two ideas, which can be used with either Contiki or a plain multithreading operating system seating on top of the control layer.

The remainder of the paper is structured as follows: in Section 2, we briefly present the technical background for the definition of the control layer; Section 3 gives the hardware platform example; Sections 4 and 5 describe the principles of our approach, and then suggest extensions; Section 6 describes the actual implementation; Section 7 provides an evaluation of the whole approach; Sections 8 and 9 review related work and conclude.

2. Background on Synchronous Languages

The essential points of synchronous languages semantics [4] can be explained with synchronous products of Boolean Mealy machines (BMMs), as described in [20]. In such machines *inputs* and *outputs* are distinguished, and the communication is based on the asymmetric *synchronous broadcast* mechanism.

Figure 1 is an example. Machine Sa (resp. Sb) reads a (resp. b), and emits a b (resp. c) every two a's (resp. b's). SE is the result of their composition. It is a machine that reads a and emits a c every four a's. Notice that emitting b in Sa, and reacting to b in Sb, are combined into a single transition, making communication instantaneous.

In all synchronous languages, a program is made of several components that can be viewed as separate BMMs. Several constructs allow to combine them, in parallel or hierarchically. The various machines communicate via the synchronous broadcast, by sending and receiving signals. From a parallel program, the compilers produce a piece of code called the *reactive kernel*. This kernel has to be wrapped in some loop code, which calls the kernel repeatedly, to make it execute one transition at a time. The C code of the reactive kernel is sequential: the parallelism present in the original program has been *compiled*, *i.e.,* statically scheduled. Note also that the size of the resulting code is linear in the size of the original BMMs, not in the size of their product.

For the example given above, we can encode the two automata into Lustre/SCADE [12], and then use a compiler from Lustre to C code. The code produced looks like the one on Listing 1. The first part is the "reactive kernel" produced by the compiler: a *step* function, and an *init* function, plus the declaration of the state variables. The second part shows how to use such a kernel, by providing an output function, and calling the kernel in an infinite loop that provides it with inputs. The execution of this code outputs "1" every 4 occurrences of value 1 for the input a.

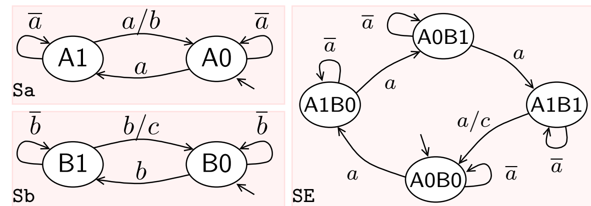

Figure 1. Two Boolean Mealy machines Sa, Sb synchronized via a signal b, and the result of their composition SE.

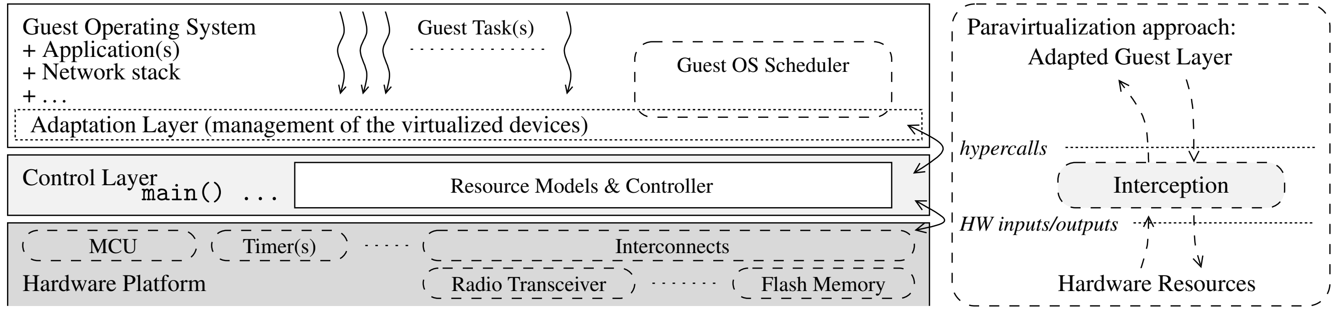

Figure 3. Global description of the approach.

```
// the reactive kernel produced by the compiler:
int M6, M13, M4;              // state variables
void init () { M4 = 1; } // initialization
void run_step (int a) {
  int L1, L2, L3, L8, L10, L12, L15;
  L3 = M4 | M6; L2 = ~L3; L12 = M4 | M13;
  L10 = ~L12 & a; L1 = L2 & L10; main_O_c(L1);
  L8 = L3 & ~L10; L15 = L12 & ~a;
  M6 = L8 | L1; M13 = L15 | L10; M4 = 0; }
// to be added to get a main program:
// output procedure:
void main_O_c (int x) { printf ("%d\n", x); }
int main () {
  int a; init ();              // initialization
  while (1) {                  // infinite loop
    printf ("Give a (0/1): "); // get inputs
    scanf ("%d", &a);
    // compute next state and produce outputs
    run_step (a); } }
```

Listing 1. Reactive kernel obtained by compiling the example of Figure 1, and an example main program using this kernel.

3. Example Platform

Figure 2 is a block diagram describing the Wsn430 hardware mote for wireless sensor networks. It is composed of an MSP430 microcontroller including several on-board peripherals (timers, ADCs, USARTs — universal synchronous/asynchronous receiver/transmitter, etc.), a CC1100 radio transceiver, a flash memory module and various sensors. A network simulator, along with a cycle accurate emulator can be used in order to test and debug applications and full systems from their target binary code [14]. Concerning shared resources, one can note that the flash memory module and the external RS232 serial link share the same USART module of the MCU. Therefore, we need to avoid simultaneous accesses to these resources.

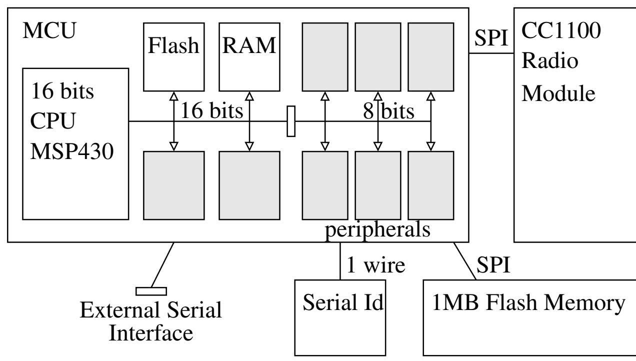

Figure 2. The Wsn430 mote (taken from [14]).

4. Principles of the Solution

The solution we propose can be explained by looking at the implementation we obtain for the example given in Section 3. For sake of simplicity, some tricky points are not explained on this example (more details are provided in Section 5).

In the sequel, we call *tasks* (or sometimes *guest tasks*) the execution flows executing concurrently in the guest operating system (OS) and application layer, whatever the concurrency model of this system is. For instance, a task can be a classical thread or a protothread [11] (as the basis in the Contiki OS).

4.1 Main Structure

Figure 3 describes the main structure. From bottom to top, it shows: the hardware, the *Control Layer* (CL), and the guest code (the guest operating system, plus the application code). In order to communicate with the hardware, the guest uses dedicated *function calls* instead of direct low-level register operations. The element called *adaptation layer* on Figure 3 represents this modification. Modulo this slight modification of the hardware accesses, any existing OS can be ported on top of the CL.

The CL is the key element. It implements the global control objectives for resource and power management by intercepting hardware requests (*i.e.,* interrupt requests) and software requests (from the guest, via the adaptation layer). It maintains an up-to-date view of the current states of all the hardware resources.

The CL presents a simplified view of the real hardware to the guest, by exporting a set of functions that may be called by the guest through the adaptation layer. These functions play the same role as the *hypercalls* of the paravirtualization approaches [24].

4.2 The Adaptation Layer

The adaptation layer is the part of the guest operating system that needs to be modified in order to be executed with the CL. It mainly comprises a set of simple functions issuing software requests to the underlying layer by using so-called hypercalls. It can also register *callbacks* to be executed upon emission of a given event by the CL.

```
1  turn_adc_on ()
2   if (on_sw (adc_on) = ack_a) return success;
3   timer_wait (some time); // Consider we can
4   turn_adc_on ();         // try again later
```

Listing 2. Function of an ADC driver.

Listing 2 illustrates a function for an ADC driver. *adc_on* is an input of the control layer that can be refused. on_sw() is a function (hypercall) provided by the CL (see below), returning the event ack_a if the request *adc_on* has indeed been taken into account.

The adaptation layer also exposes a run_guest() function whose behavior is to schedule and execute all runnable guest tasks, if any. This function returns when all tasks are blocked and all needed

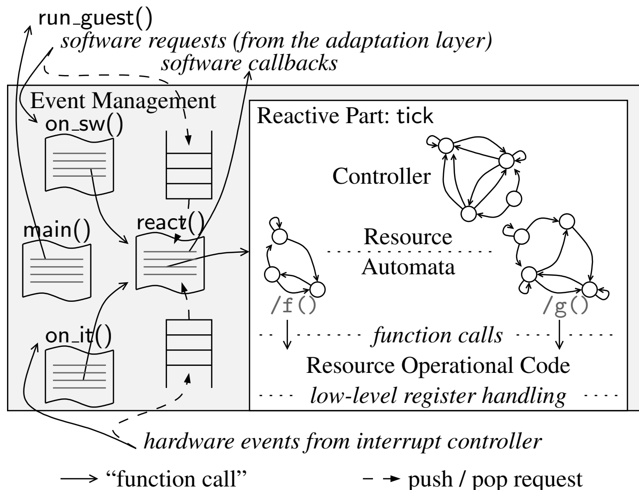

Figure 4. Details of the Control Layer.

computations have been performed by the guest, meaning that the CL can put the CPU in low power mode.

4.3 Overview of the Control Layer

4.3.1 Structure

As depicted in Figure 4, the CL is made of two parts: the *reactive part*, and the *event management* part.

The *reactive part* (also referred to as tick in the sequel) comes from all the automata of the drivers to be controlled, plus a controller. These drivers consist in BMMs as depicted in Section 2, that produce outputs triggering the execution of *resource operational code* (essentially low-level code accessing registers of the devices). They also produce *output events* that provide information about the state of the resources to the adaptation layer. Inputs of these BMMs are twofold: requests emitted by the adaptation layer that trigger operations on the devices, plus *approval* signals that allow the controller to restrict this behavior (details on approval signals and the controller are given in Section 4.5).

The tick is the reactive kernel obtained from the compilation of the controller and the resource automata, when composed in parallel (as mentioned in Section 2). The tick is entirely passive: it has to be called from the other part of the CL (see below); when called (using the tick.run_step() function), it executes exactly *one* transition of the compiled automata, then executes some low-level code, and produces *output events*.

The *event management* part is in charge of managing *queues* for hardware and software requests, and building *input events* in order to call the tick. It also interprets the *output events* produced by the reactive part in order to send information to the upper layers. The *event management* part is made of the two queues, plus several pieces of code. The hardware event queue is filled by the hardware only; the software event queue is filled by the guest layer only. We first describe each piece of software. The complete behavior that results from their organization is best understood by looking at the two possible execution paths of Figure 5. The pieces of software are as follows:

- on_it() is the interrupt handler: its execution is launched by the occurrence of some interrupt (which has also posted an element in the hardware queue); it calls react();
- on_sw() is executed by the adaptation layer so as to emit a software event to be pushed in the software queue; it also calls react();
- react() consumes the elements in the two queues, in order to build an *input event* to be given to tick. It is a loop, calling tick.run_step() until the two queues are empty; when a software request s is taken from the queue, and used as part of an event to run one tick, react() is able to interpret the *outputs* of the tick execution, and to transmit information to the adaptation layer that had called on_sw(s).

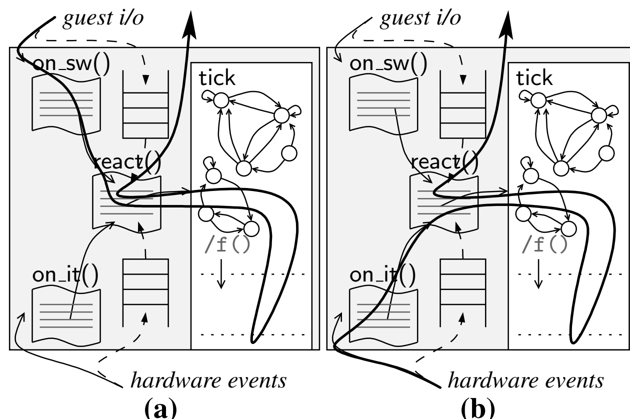

Figure 5. Execution Paths in the Control Layer, either triggered by the emission of a software request (a) or a hardware event (b).

4.3.2 Example Execution Paths

Figure 5 describes the execution paths corresponding to software and hardware requests alone. Figure 10 and Section 4.6 illustrate a more general execution, where hardware and software requests may happen concurrently.

Let us look at Figure 5-(**a**) first, and suppose that no hardware interrupt occurs. The guest code needs to perform an operation on the hardware (*e.g.,* turn the ADC on); to do so, it calls a function turnADCon() of the adaptation layer. This function posts a software request *adc_on*, by calling the on_sw() function. Then, on_sw() immediately calls react(), that picks an element in the software queue (the request just posted), and calls tick.run_step() with an event of the form: *adc_on* $.\,\overline{x}\;.\,\overline{y}\ldots$ where $x, y, \ldots$ are the other software and hardware events. The execution of the tick with such an event executes the appropriate transition from its current state, may execute some low-level code, and returns by providing the *output events* of the transition. react() analyzes this output, and may call some *callback* function of the adaptation layer, to report about what happened when the software event was treated. In this execution path, the flow of control is not stopped during the treatment of one software request; as a result, the loop in react() iterates exactly once for each request.

Let us look at Figure 5-(**b**). A hardware interrupt occurs, *i.e.,* the hardware puts an event irq_a in the hardware queue, and then "calls" on_it(). This may happen at any time, in particular while the processor is busy executing the software, including one call of react(). If a call to react() is executing currently, on_it() leaves the interrupt in the hardware queue, but does nothing to treat it (see below). If no call to react() is executing currently, then the software is interrupted, and on_it() calls react(). That consumes the hardware event just posted in the hardware queue, and calls tick.run_step() with an event of the form: $irq_a\;.\,\overline{x}\;.\,\overline{y}\ldots$ where $x, y, \ldots$ are the software events, and the other hardware events. The execution of the tick with such an event executes the appropriate transition from its current state, may execute some low-level code, and returns by providing the *output events* of the transition. react() analyzes this output, and may call some functions of the adaptation layer. For instance, in case of an interrupt from the timer, the adaptation layer may have to wake up some task in the guest code.

```
1  main ()
2   tick.init (); // Initialize the reactive part
3   while (true)
4     // Run guest until it requests entering LPM
5     // by returning from this function:
6     run_guest ();
7     // Enter LPM: this function will return upon
8     // the next hardware event occurrence (cf.
9     // function 'on_it()' of Listing 4).
10    enter_lpm ();
```

Listing 3. The main() Function.

```
1  on_it ()
2   // Notice the event has already been pushed in
3   // hw_queue, and interrupts are disabled
4   if (! already_in_reaction)
5     react ();     // Trigger reaction
6     wake_up (); // Then wake up the CPU if it
7                 // was in LPM hitherto
```

Listing 4. The on_it() Function.

4.4 Details on the Event-Management Part

Let us describe the left part of Figure 4. The goal of this part is to efficiently interleave executions of the reactive part along with the guest operating system. It also manages all input requests, translating them into input events to be given to the reactive part, and conveys output information to the adaptation layer.

The *events* triggering executions of the reactive part are:

- *hardware events* (or *interrupts*), notifying a device operating mode transition. For instance, these information can be low-level timer counter expiration, general purpose digital input state update, etc.
- *software requests*, notifying requests from the guest operating system.

4.4.1 Running the Guest Operating System

As usual, the main() function presented in Listing 3 is the entry point of the whole system and application code. After having initialized the tick code and the guest software using tick.init(), it enters in an endless loop. It then continuously activates the execution of the guest task(s) through the provided run_guest() function (typically, a call to the guest task manager, meaning that it runs until all tasks are blocked). If the latter function returns, then no more task is runnable; *i.e.,* the guest would have put the CPU in low power mode (LPM) if it were executing on bare hardware. The main() function can then put the CPU into LPM. It benefits from the full knowledge of the reactive part about the states of the hardware devices in order to compute the best admissible LPM (*i.e.,* in order to ensure its wakening by a hardware event). The guest tasks are permanently rescheduled and executed through the run_guest() function upon the occurrence of a hardware event.

4.4.2 Handling Input Requests

The on_it() function depicted in Listing 4 is called by low-level interrupt handlers, with interrupts being disabled, upon insertion of a new hardware event into the associated queue. It calls the react() function if it was not already running when the interrupt occurred (with the help of the already_in_reaction flag). It eventually wakes up the CPU when needed.

The role of the on_sw() function (*cf.* Listing 5) is to queue new software requests and trigger reactions upon their emission by the adaptation layer. It returns the corresponding output of the reactive part.

The react() function in Listing 6 behaves as follows: when called, it sets the already_in_reaction flag so that upcoming hardware events do not trigger reactions themselves (*cf.* on_it() function in Listing 4). This flag is reset at the very end of the reaction.

```
1  on_sw (input_signal)
2   // Create a new request from the input:
3   sw_req.create (input_signal);
4   // Disable interrupts to protect requests
5   // management:
6   disable_interrupts ();
7
8   // Push the event in the software requests:
9   sw_queue.push (sw_req);
10  react ();              // Trigger reaction
11  enable_interrupts (); // Re-enable interrupts
12
13  // Return the result that has been recorded by
14  // the 'react()' function
15  return sw_req.get_result ();
```

Listing 5. The on_sw() Function.

```
1  react ()
2   // Announce we start the reaction:
3   already_in_reaction = true;
4   all_outputs.empty ();
5
6   do {
7     // Build an input event by extracting the
8     // software request (if any) and hardware
9     // events from the two queues:
10    <input_event, sw_req> =
11      build_input_event (hw_queue, sw_queue);
12
13    // Notice enabling interrupts allows new
14    // hardware events to be pushed into hw_queue
15    enable_interrupts ();
16    outputs = tick.run_step (input_event);
17    disable_interrupts ();
18
19    all_outputs.merge (outputs);
20    // If there was a software request, then
21    // setup its result:
22    if (sw_req) sw_req.set_result (outputs);
23
24  } while (! hw_queue.is_empty () ||
25           ! sw_queue.is_empty ());
26  // Here, both queue are empty.
27
28  // Eventually, launch callbacks associated with
29  // all emitted outputs (if any):
30  launch_callbacks (all_outputs);
31  // Notify we leave the reaction:
32  already_in_reaction = false;
```

Listing 6. The react() Function.

The loop from lines 6 to 25 extracts and treats all requests from both the software and hardware event queues until they become empty. Each turn, all requests are popped from the queues and an input event is built and given to the reactive part (on line 16).

The output of this execution step of the tick is set as result of the software request, if any, that triggered it (line 22).

Again, this result is merged with all outputs gathered during the preceding executions of the loop (in set all_outputs, initialized to the empty set at the beginning of the function). This set will serve on line 30 to launch all previously registered callbacks associated with the outputs that have been emitted during the reaction.

4.5 Details of the Reactive Part

Let us now look at the details of the reactive part of Figure 4. This component gathers all the (possibly controllable) Mealy machines

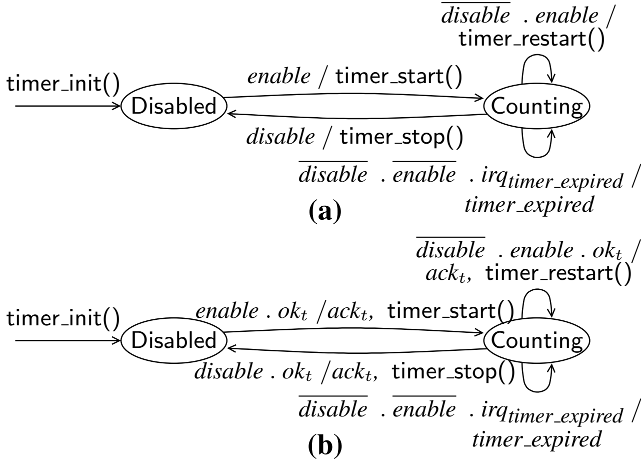

Figure 6. Original (a) and Controllable (b) Device Driver Automata for a Timer.

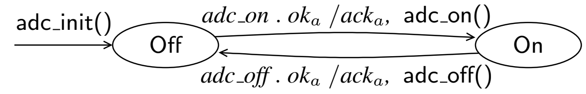

Figure 7. Controllable ADC Automaton.

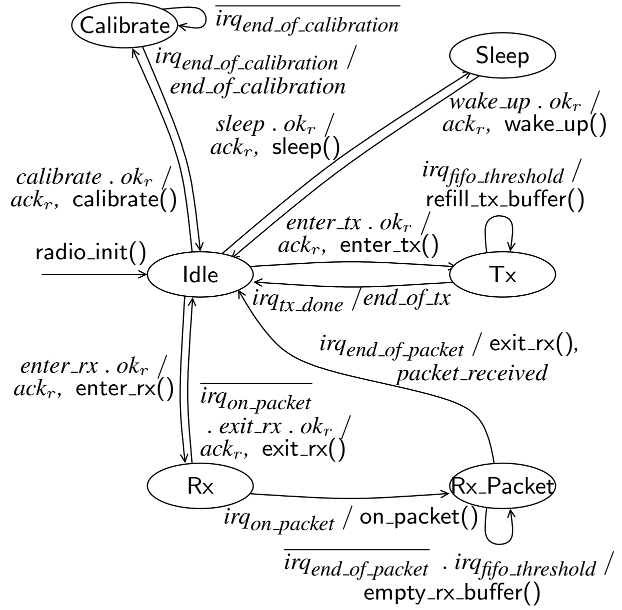

Figure 8. Controllable Radio Transceiver Automaton.

modeling the hardware devices along with the associated controller. All these components are compiled together into a piece of sequential code according to the synchronous parallel composition paradigm (*cf.* Section 2).

The inputs of the automata are Boolean formula built from the hardware interrupts (*e.g.,* a timer interrupt) and the software requests (*e.g.,* turn the ADC on). The outputs are acknowledgments (*e.g.,* the request for turning the ADC on has actually been taken into account) and several signals that abstract the events occurring at the hardware level (*e.g.,* the timer has expired).

4.5.1 Device Driver Machines

Designing the control path of a usual driver as an explicit automaton is quite natural. The states of this automaton reflect the operating modes of the device. The transitions are triggered by inputs that may be of two kinds: requests for changing modes, and hardware events. The idea is to guarantee that the state of the automaton always reflects the state of the device. The outputs on the transitions may represent low-level code (accesses to the device registers) that has to be executed to perform the real device operation.

In the sequel, we use the following syntax for a transition label: "$i_1 . \overline{i_2}$ / o_1, o_2, f()" where $i_1 . \overline{i_2}$ is an example Boolean formula built from the set of software and hardware inputs, o_1 and o_2 are example outputs, and f() is the call to some low-level code. For the sake of simplicity, self-loops with no outputs are not represented on the figures.

An important point here is the notion of *controllability*. Indeed, if we want to meet global control objectives, the requests from the software (and potentially some of the hardware events) should not always be accepted by the device driver. In the vocabulary of controller synthesis, it means that the automata to be controlled should have *controllable inputs*; if they do not have enough controllable transitions, then the global control objective may be unfeasible. In the sequel, we describe controllable automata for the devices. For one of them (the timer), we explain how to transform a non-controllable device driver into a controllable one.

Note that for the sake of clarity, we explain the notion of controllability using a timer: it is a simple example of small device involving controllable and non-controllable inputs. However, in practice, one would prefer leaving at least one timer non-controllable so as to avoid latency problems when enabling the timer.

Figure 6-(**a**) is the automaton for a timer driver. timer_init(), timer_start(), timer_restart() and timer_stop() are low-level functions updating the timer operating mode and registers. The input $irq_{timer_expired}$ is a hardware signal, whose meaning is the expiration of this timer. Finally, *disable* and *enable* are input requests issued by the upper software layer to drive this device, and *timer_expired* is an output signal reflecting the expiration of the timer.

Modifying this device driver so that it becomes *controllable* is done by introducing additional inputs. In Figure 6-(**b**), the automaton of Figure 6-(**a**) has been modified by introducing an approval input (ok_t) meaning that a transition is triggered only when both the input request and the approval signal hold. A controller can then inhibit the controllable transitions of a resource by not emitting the associated authorization. The condition $enable . \overline{ok_t}$ occurring in state Disabled is part of the implicit "else" loop; which corresponds to the fact that the request *enable* has been refused.

In order to notify the requesting software that a state transition has not been approved, additional outputs are also used (ack_t in the timer example). They are emitted when controllable transitions are permitted. Note also that in this case (and in the sequel of this paper), these new signals are only inhibitors (*i.e.,* they are used by the controller to prevent some state transitions from occurring), but one could also augment behaviors by adding other kind of inputs.

Figure 8 describes a slightly simplified radio transceiver driver automaton (without error handling transitions and wake on radio feature). This automaton is used in the sequel.

4.5.2 The Controller Automaton

Given the full set of controllable device automata and some global properties to be enforced, a controller can be designed.

Figure 9 is an example of a controller designed from the radio transceiver and ADC automata described in Figures 8 and 7 respectively (it only controls these two devices for clarity of the example). This controller ensures the exclusiveness between three energy-greedy states of the former and the operating mode of the

Figure 10. Example of execution.

Figure 9. Basic Controller Automaton.

latter, when put in parallel with them. Indeed, neither the Rx, Tx nor Rx_Packet are reachable from all the other states of the radio driver until ok_r holds. Again, this internal input is never emitted by the controller when a transition leading to the Adc state is triggered. Conversely, the On state of the ADC is not reachable when the radio is in one of its highly consuming modes.

4.5.3 The Compiled Automaton

Finally, the drivers automata and the controller are compiled into a single piece of code that forms the tick, and behaves as the product automaton. For instance, and going back to our basic controller example, this automaton will be in state Free_Off_Idle when the Controller, ADC and Radio will be in Free, Off and Idle states respectively. This global state leads to another state Adc_On_Idle through a transition with condition $\overline{enter_tx} \,.\, \overline{enter_rx} \,.\, \overline{calibrate} \,.\, \overline{sleep} \,.\, adc_on$ and outputs ack_a, adc_on().

4.6 A Complete Example Execution

We illustrate on Figure 10 a complete interleaving of executions between the CL and the guest layer for a simple example application, and a reactive part comprising device automata of Figures 6-(**b**), 7 and 8, along with the controller of Figure 9. The time goes from left to right. For sake of simplicity, we trace the states of the controller, the ADC and the radio transceiver between reactions. We also depict the role of the adaptation layer in this process by describing the software requests emitted to the CL.

The guest layer is composed of two tasks; one performs the sensor management by using the ADC ($\mathsf{T}_{sensing}$), and the other periodically operates on the radio network ($\mathsf{T}_{networking}$). Executions of these two tasks are triggered by two distinct timers. Again, consider the CPU is initially in one of its low power modes.

First, the hardware event $irq_{timer_1_expired}$ occurs, and triggers an execution of the reactive part. This outputs the $timer_1_expired$ signal, and the react() function launches the associated callback. This callback restarts the $\mathsf{T}_{sensing}$ task execution, returns, and then the guest layer can continue its execution (the main() function calls guest_run()).

Upon emission of a software request by this task (a call to on_sw(adc_on)), a new reaction is triggered. Since the complete tick automaton is in state Free_Off_Idle at this time, this input triggers the transition taken as example in Section 4.5.3: the request is then approved and it turns the ADC on (the low-level adc_on() function is called). The request returns ack_a, and the task stops its execution: the main() function can then call enter_lpm().

Ibid, when $irq_{timer_2_expired}$ occurs and wakes up the CPU, the task $\mathsf{T}_{networking}$ is scheduled. While computing, this task is interrupted by an $irq_{timer_1_expired}$ hardware event, a reaction, and then a new call to the callback associated with $timer_1_expired$. In turn, this callback enables a new execution of the sensing task. When exiting the interrupt handler, the networking task continues its execution, and tries to put the radio transceiver in its emitting mode (on_sw($enter_tx$)). While the controller is in state Adc, it does not allow this software request, and its result is $\overline{ack_r}$ so the networking task can choose to yield and retry later.

Afterward, the sensing task executes and turns the ADC off (on_sw(adc_off)); the associated reaction executes the adc_off() function, and updates the state of the controller. Eventually, the next emission of $enter_tx$ is permitted by the reaction part, and the emission can succeed.

5. Extensions

The general approach we have just described can easily be extended to treat more complex cases. We explain some of them below.

Allowing direct accesses to the hardware: In some cases, we may need to allow direct access to the hardware resources, for some of the guest tasks. For instance, if we use a multithreading OS as the guest, assume the existence of a guest task that prints reports using an UART connected to the bus, directly; the task continuously sends characters to the device, hence using the bus. Suppose now that another device uses the same bus (for instance, the flash memory). We need some control for bus accesses, but since the task has a direct access to the hardware, it seems this cannot be done with our approach. In fact, it can be done by considering the task as an additional object similar to a device. We model its behavior with a two-state automaton (using the bus, or not) and controllable transitions, and we add this automaton to the set of automata for which we have to design a global controller. Then, we need to make sure that this two-state automaton always reflects the real state of the task. To do so, we implement some communication between the controller in the CL and the scheduler of the guest system. When the controller forbids a transition from the state "*not using the bus*" to the state "*using the bus*" of the task model, it also communicates with the guest scheduler, requesting him to remove the task from the list of eligible tasks. This mechanism is such that a global invariant is maintained: whenever the task is running (and accessing the bus), the CL is in state "*using the bus*", which prevents other devices from using the bus.

Best low-power mode of the MCU: Our approach can also be extended to force the CPU to be in the best low-power mode as possible, considering the possibilities for being woken up. Let us take the MSP430 as an example. This micro-controller has 6 operating modes, among which: the *Active* mode, in which everything is active (this is the mode with the highest consumption); the *LPM4* mode (the one with the lowest consumption), in which there is no RAM retention, the real-time clock is disabled, and the only way to wake up is by an external interrupt; the *LPM3* mode (having an intermediate consumption) in which there is only one peripheral clock available; the MCU can be woken up by a timer, and by external interrupts.

Ensuring that the MCU is always in the lowest consumption mode, and yet can be woken up, is an instance of a global control problem. Indeed, the MCU should not be put in its lowest consumption mode, from which only an external interrupt can exit, if there is no chance for such an external interrupt to happen. Information on which external interrupts may occur is given by the *states* of the automata that model the hardware devices. We have to model the MCU by an automaton in which all the available mode changes are represented by controllable transitions. The "*best low-power mode*" objective can then be stated as an invariant property (avoid some global states), plus a *quality* objective that can be taken into account when designing the controller.

6. Implementation

In order to show that our proposal is realistic, we have implemented the CL on top of the hardware architecture described in Section 3. We have tested it using the cycle-accurate platform and network simulator provided with the Worldsens tools [14].

Implementation of the Reactive Part: According to what has been described in Section 2, we have implemented a reasonable and usable set of device drivers so as to build up a working CL on top of the Wsn430 hardware platform. The reactive part has been implemented using the SCADE industrial tool-chain [12].

We have manually designed a controller for all the devices and resources of the platform. It ensures simple safety properties like state exclusions for shared resource management (*e.g.,* the flash memory and RS232 link resources use the same serial module with distinct modes on the Wsn430 platform), as well as reduction of current consumption peaks by avoiding reachability of global states when two or more peripherals (such as ADC and radio transceiver) are in their highest consumption modes.

Guest Layers: Regarding the guest layer, we have ported two operating systems that could also run on the bare hardware, onto our CL implementation.

Targeting Contiki: The adaptation of Contiki [10] required writing a set of device drivers dedicated to the abstract hardware exposed by the CL. The writing of these drivers is easy, whatsoever the strategy for handling rejected requests is: one could choose to retry requests after a given time, or return a dedicated error code to let the application processes select a more suitable strategy.

Targeting a Multithreading Operating System: The second operating system ported onto the CL is a priority-based preemptive multithreading kernel we designed from scratch[1]. Writing adapted device drivers in this guest was similar to writing those of Contiki, except for the task management and synchronization parts, as a result of the change of concurrency model.

7. Discussion and Evaluation

The technical elements we have described constitute a complete proof of concept, for the implementation of centralized resource control policies in a paravirtualization framework. The implementation runs on top of a quite detailed emulator, which is a reasonable guarantee that it will also work on the real hardware. However, providing global property enforcement necessarily introduces some computation and memory overhead. In order to show that the proposed solution is practicable, we estimate this overhead compared to available data about existing OSes for WSNs.

Evaluation of our solution: The memory footprint of the `tick` is about 1.5 to 2 KB (recall that the size of the `tick` is linear in the size of the individual automata descriptions, as stated in Section 2). Stack space required for its computation is about 100 bytes, but it could be shared among threads in case of multithreading guest, since there is always at most one guest task running the `react()` function at a time. Other parts of the CL mainly consist in the low-level code of the device drivers that would be in the guest otherwise. Rough implementation of the code and static data structures related to the interaction between this low-level code, the guest, and the `tick` occupies 1 KB.

The time overhead involved by running one step of the compiled automata of the `tick` depends on the input events and requests: it takes at most 1,600 cycles ($200\mu s$ on an MSP430 clocked at 8MHz) in our current implementation.

	TinyOS-1.1 (kernel)	TinyOS-1.1	RETOS	MtK/CL
ROM	11.2 KB	21 KB	23.1 KB	24 KB
RAM	311 B	798 B	824 B	806 B

Table 1. Memory footprint of some existing OSes for WSNs, with various sensor drivers and network modules (data taken from [6]). The "MtK/CL" column represents ROM footprint of our multithreading kernel and CL implementation.

Comparison with existing solutions: Due to the lack of reference figures regarding memory overhead and computation time involved by existing solutions for global control in WSN OSes, we evaluate the overhead of our solution by comparing it to solutions without such control. We sum up in Table 1 the typical memory footprint

[1] Actually, we first tried to design this operating system having our device driver model in mind. Observations about the paravirtualized nature of it came later.

of some already existing OSes for WSNs. Compared to these results, our solution involves a code size increase of less than 10%. This memory overhead is very realistic *w.r.t.* the benefits of global resource control that our approach can manage.

ICEM with n arbiters & m power managers	tick
$\lesssim 350n + 400m$	$\approx 1,600$

Table 2. Typical cycle overhead of ICEM decentralized shared resource arbiters and device power managers, compared to our global control implementation.

Table 2 compares cycle overheads of decentralized control in the ICEM framework [18] and our proposal. The overhead introduced by our solution for global control and power management is reasonable, even if the tick is run each time a resource state change can occur.

Discussion: The results show that our solution for global control is very practical, even though it involves an overhead, both in terms of memory usage and computation cycles.

However, when time and memory really matter, one could take advantage of the synchronous nature of the tick so as to implement it efficiently in a dedicated hardware module. Other ways to improve these results would be to design a dedicated compilation scheme of synchronous programs for memory constrained systems. Guest operating systems code could also be further reduced by taking advantage of the properties enforced by an underlying CL.

Furthermore, developing a device driver for the CL having the associated automaton in mind, reveals itself simpler than for classical OSes. Such development is easier using our approach, where a clear distinction is made between low-level code that affects device operating modes, and effective application code.

8. Related Work

Operating Systems for WSNs: Whereas originally designed for classical real-time embedded systems, MANTIS [5] has been ported to some motes. It is a priority-based preemptive multithreaded operating system and has shown the power of this concurrency model for wireless sensor networks by enabling implementation of lengthy tasks. RETOS [6] and Nano-RK [13] are also multithreaded operating systems. However, neither MANTIS nor RETOS provide support for global resource management. Nano-RK provides static reservation mechanisms for energy and timing management of applications.

TinyOS [17] is the most widely known operating system for WSNs. It is fully component-based, event-driven and based on the NesC language [15], thus facilitating composition and reuse of previously written code. Component connections express the overall structure of the operating system along with the application. Nevertheless, building complex applications exploiting a broad range of the available devices supplied by a mote implies using dozens of components, hence leading to component bindings and interactions that are hard to apprehend globally.

Energy management for WSN nodes: Several works already addressed the problem of energy management within a WSN node by means of a dedicated device driver architecture. We briefly describe here the ones that are close to our proposal.

[18] presents the solution promoted within TinyOS. It consists in addressing both concurrency and energy requirements in a single framework called ICEM, a core component of TinyOS 2.0. The key idea behind ICEM is to offer a driver interface based on (potentially concurrent) application I/O requests to better control devices power states. From the application point of view, this concurrency level can be expressed by means of distinct driver classes. *Virtualized drivers* allow implicit concurrency between multiple users. Client requests are buffered and scheduled according to some desired properties (*e.g.,* fairness), and a per-client state is maintained to control the device power state. *Shared drivers* also support multiple users, but they offer a lower level of interface in terms of *(power) locks*: each client should acquire a lock before using a shared driver. A special component, the power manager, is responsible for implementing the energy management policy of a shared driver (*e.g.,* powering off this driver as soon as its associated lock is idle). ICEM's architecture leads to a decentralized energy and resource management scheme, split inside each driver class.

In [9], Choi et *al.* suggested a global device driver architecture dedicated to multithreaded sensor network operating systems. This architecture provides also three kinds of driver models (offering several trade-offs between performance and complexity), and some global operating system services to control shared access and energy consumption through a so-called device manager. This control is performed by means of a fixed set of dedicated request functions (either non blocking, or with a specified waiting time). Thanks to a centralized data structure indicating the current state of each device (and some specific device control functions) the device manager can assign the best suitable low-power state to the MCU and each hardware elements.

Although this proposal is closed to our work in the sense that it offers a global control, it suffers from some drawbacks and limitations. The multithreaded device manager certainly becomes harder to write as soon as the number of devices grows. Liveness problems may also occur, and not all devices are controlled (*e.g.,* the timers are left uncontrolled, however, they can impact the best low-power mode).

Other approaches has been proposed where the application logic is not involved in resource management decisions. ECOSystem [25] is a general purpose operating system that integrates an explicit notion of "energy resource" into the scheduling mechanism of shared system devices. Eon [22] can be viewed as an *energy-aware* data-flow programming language, where flow paths within programs are annotated with energy states by the programmer. Then, at runtime, Eon "adapts" the application code by selecting a suitable dataflow path according to current energy availability.

Pixie OS [19] is a more recent operating system dedicated to sensor nodes. It borrows some ideas from ECOSystem and Eon. Its purpose is to enable some *resource-aware programming model* with respect to energy, radio bandwidth, storage, etc. It relies on a dataflow model plus a notion of *resource tickets* to abstract the allocation of physical resources. Resource management policies can be enforced by means of (dedicated) resource brokers that deliver resource tickets to the application. The notion of "broker" is rather close to the global controller we propose. However, it differs in several points: first, brokers are dedicated to specific resources, meaning that a broker competition could be necessary to enforce global properties (related to several resources); second, correctly estimating an energy quantum for a given work unit could be a difficult task, and a too conservative approach could degrade the node performance; finally, there is no general technique for designing a "correct" broker with respect to a given policy (like the automated control approach we propose).

Automated control and operating systems: The research community on computing systems, in particular operating and distributed systems, has been showing interest for the use of control theory for some years now. [1] is a very good introduction to the field, and exposes several applications, among which power control.

Formal models for driver design: In [23], formal models of drivers are used as an abstract specification, from which the code

can be produced. The whole approach is comparable to our use of automata labeled by function calls for the low-level programs.

In [8], formal models are used to *map* a MAC algorithm on top of a complex radio device; the radio device has more states than the functional states that matter for the software. The proposed method may allow, for instance, to specify in the MAC algorithm that the radio should go from idle to transmit mode; the formal model of the radio device is then used to transform this functional behavior into a more complex behavior of the radio, which needs to go through various states of different energy levels between idle and transmit. The mapping algorithm can probably be formulated as a control objective, with controlled events forcing transitions instead of inhibiting them, but this would need further investigation.

9. Conclusions and Further Work

We proposed a software architecture for embedded systems, allowing for global control of the hardware devices. It requires a slight adaptation of the application software, but does not change the way it is designed and programmed. As an example, we demonstrated the use of Contiki and a multithreading operating system on top of our software architecture.

The advantages of the approach are: (i) a clear expression of the global control objectives, that helps designing the global controller; (ii) the use of a synchronous language for the control layer, which makes it possible to compile the set of drivers and the controller into a piece of statically-scheduled efficient code; (iii) the easy extensibility of an existing control layer.

Further work will be devoted to several classes of extensions. We will adapt widespread operating systems such as TinyOS or MANTIS, so as to be able to run the large amount of already written applications, on top of our control layer. We will also look at situations in which the applications may need to *book* some resources in advance, to prevent their requests from being canceled; this can be done with more complex automata for the resources, but essentially the same kind of controller as presented in this paper.

Finally, another branch of research would be to implement the whole software of a node in some synchronous language, and to perform static analysis and then static scheduling. When the application is written using event-driven programming, as in TinyOS for instance, it would not change the practice a lot to write it in the kind of automaton-based language we used here (the automata exchange signals in a synchronous way to express the synchronization, and this mechanism is indeed compiled; their transitions are also labeled by calls to pure C code, allowing for an easy reuse of, *e.g.,* the protocol stack). This would provide both: *structured event-driven programming* with no additional runtime cost, and a formal model of the whole system that can be analyzed before it is deployed.

References

[1] T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu. *Introduction to Control Theory And Its Application to Computing Systems*. June 16 2009.

[2] T. F. Abdelzaher, L. J. Guibas, and M. Welsh, editors. *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN 2007, Cambridge, Massachusetts, USA, April 25-27, 2007*. ACM, 2007.

[3] K. Altisen, A. Clodic, F. Maraninchi, and É. Rutten. Using Controller-Synthesis Techniques to Build Property-Enforcing Layers. In P. Degano, editor, *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2003.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[5] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *MONET*, 10(4):563–579, 2005.

[6] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In Abdelzaher et al. [2], pages 148–157.

[7] V. Chandra, Z. Huang, and R. Kumar. Automated control synthesis for an assembly line using discrete event system control theory. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 33(2):284–289, 2003.

[8] A. Chis, E. Fleury, and A. Fraboulet. An optimized MAC layer to physical device mapping methodology. In *Mobility Conference*. ACM, 2009.

[9] H. Choi, C. Yoon, and H. Cha. Device Driver Abstraction for Multithreaded Sensor Network Operating Systems. In R. Verdone, editor, *EWSN*, volume 4913 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2008.

[10] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN*, pages 455–462. IEEE Computer Society, 2004.

[11] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In A. T. Campbell, P. Bonnet, and J. S. Heidemann, editors, *SenSys*, pages 29–42. ACM, 2006.

[12] Esterel Technologies Ltd. SCADE language reference manual. `http://www.esterel-technologies.com/`.

[13] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *RTSS*, pages 256–265. IEEE Computer Society, 2005.

[14] A. Fraboulet, G. Chelius, and E. Fleury. Worldsens: development and prototyping tools for application specific wireless sensors networks. In Abdelzaher et al. [2], pages 176–185.

[15] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11. ACM, 2003.

[16] A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190–225, 2009.

[17] J. L. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS*, pages 93–104, 2000.

[18] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. E. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 251–264. ACM, 2007.

[19] K. Lorincz, B. rong Chen, J. Waterman, G. W. Allen, and M. Welsh. Resource aware programming in the Pixie OS. In T. F. Abdelzaher, M. Martonosi, and A. Wolisz, editors, *SenSys*, pages 211–224. ACM, 2008.

[20] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Comput. Lang.*, 27(1/3):61–92, 2001.

[21] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, Jan. 1989.

[22] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In S. Jha, editor, *SenSys*, pages 161–174. ACM, 2007.

[23] S. Wang and S. Malik. Synthesizing operating system based device drivers in embedded systems. In R. Gupta, Y. Nakamura, A. Orailoglu, and P. H. Chou, editors, *CODES+ISSS*, pages 37–44. ACM, 2003.

[24] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *OSDI*, 2002.

[25] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: managing energy as a first class operating system resource. In *ASPLOS*, pages 123–132, 2002.