# Operating System Support for Handling Heterogeneity in Wireless Sensor Networks[*]

Lucas Francisco Wanner, Arliones Stevert Hoeller Junior,
Fauze Valério Polpeta and Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration
Federal University of Santa Catarina
PO Box 476 – 88049-900 – Florianópolis, SC, Brazil
{lucas,arliones,fauze,guto}@lisha.ufsc.br

## Abstract

*Developments in Wireless Sensor Networks (WSN) hardware have led to a diversity of sensing devices, ranging from simple, micro-controlled boards to complex, highly integrated sensor-transceiver ICs. Given this diversity, the lack of proper abstraction and encapsulation mechanisms at the operating system level often forces developers to reimplement their sensing applications whenever a different sensor, radio or processor is deployed. In this paper we introduce a novel strategy for abstracting Wireless Sensor Networks Hardware, implemented for the* EPOS *operating system. It consists in providing application programmers with a high level interface for sensing components, that can latter be bound to pre-existing components that are adapted on demand at system generation time to fulfill application requirements, thus enabling programmers to code portable sensing applications in spite of the hardware diversity and without significant overhead.*

## 1 Introduction

Wireless Sensor Networks (WSN) hardware is, by its very own nature, heterogeneous and modular. Application-specific requirements drive the entire hardware design, from processing capabilities to radio bandwidth and sensor modules. Even in a same family of sensor nodes (e.g. the Berkeley *motes* family [7, 11]) one can find architectural differences that cannot be trivially abstracted. In this scenario, a sensor application developed for a given platform will seldom be portable to a different one, unless the run-time support systems on those platforms deliver mechanisms that abstract and encapsulate the sensor platform in an adequate manner. Indeed, a proper abstraction of sensor platforms becomes a key issue in the face of modern SoC (*system-on-a-chip*) solutions, which integrate micro-controller, radio and sensor modules on single ICs that can usually be configured according to user demands [2, 6, 12]. Table 1 presents characteristics of some representative WSN hardware modules in use today. Architectural differences aside, sensor modules (e.g. temperature, light, motion sensors) present an even wider range of variability. Sensor modules presenting the same functionality often vary in their access interface, operational characteristics and parameters.

A properly designed run-time support system could free application programmers from such architectural dependencies and promote application portability among different sensing platforms. Given a sensing application implemented for a certain platform, there is no strong reason why it should not be reused with another platform that fulfills its requirements (e.g. presence of a certain sensor, non-volatile memory capabilities, etc.). For instance, an operating system could deliver a temperature sensor abstraction that would be instantiated by applications giving a range and scale (e.g. linear, logarithmic, *Poisson*, user-defined, etc). The system would thus ensure proper behavior independently from the physical sensor that exists in the platform.

In the following sections we present current strategies for handling heterogeneity in *Wireless Sensor Networks* and introduce a novel model based on the *Application Oriented System Design* (AOSD) methodology [3]. AOSD introduces the concepts of *hardware mediators* and *system abstractions* that implement *scenario independent* system constructs. These abstractions are exported to the application through *inflated interfaces* and adapted according to the applications needs through *scenario adapters*. We introduce the model of *sensing abstractions* used in EPOS, an experimental *application oriented* operating system. We then show that this model is capable of properly encapsulating WSN hardware heterogeneity with virtually no overhead and producing a very small system footprint.

[*]This work was partially supported by FINEP (Financiadora de Estudos e Projetos) grant no. 01.04.0903.00.

| Mote | | | | | |
|---|---|---|---|---|---|
| Type | Rene | Mica2 | iMote | btNode | Telos |
| Year | 2000 | 2003 | 2003 | 2003 | 2004 |
| Institution | UCB | UCB | Intel | ETHZ | UCB |
| CPU | | | | | |
| $\mu$controller | AVR | AVR | ARM | AVR | MSP430 |
| Clock | 4 Mhz | 8 Mhz | 12 Mhz | 8 Mhz | 8 Mhz |
| Program Memory | 8 KB | 128 KB | 512 KB | 128 KB | 60 KB |
| RAM | 0.5 KB | 4 KB | 64 KB | 4 KB | 10 KB |
| Radio Communication | | | | | |
| Type | RFM | Chipcon | Bluetooth | Bluetooth | 802.15.4 |
| Frequency (Mhz) | 916 | 433/916 | 2400 | 2400 | 2400 |
| Rate (kbps) | 10 | 40 | 700 | 700 | 250 |

**Table 1. Typical Wireless Sensor Networks Hardware**

## 2 Related Work

Most of the current effort in development of operating systems and hardware abstraction for Wireless Sensor Networks is focused on *TinyOS* [7]. Initially developed by the University of California at Berkeley and now an open-source project maintained by several institutions, it is the most widely used OS for Wireless Sensor Networks. TinyOS was written in NesC [4], a high-level programming language that emulates the syntax and functionality of hardware description languages, and provides components for communication, thread coordination, and, of our special interest, hardware abstraction.

Nonetheless, while implementing a complete and functional hardware abstraction layer, the original TinyOS system presented a series of structural problems that hindered application portability. Each hardware platform had a complete and separate HAL implementation. While it is true that architectural differences between platforms may require separate implementations, system interfaces should remain uniform whenever possible. The original TinyOS did not provide a uniform interface for most high level system abstractions (e.g. sensors, timers, etc.), thus forcing the application programmer to understand the nuts and bolts of the underlying hardware platform and compromising application portability between platforms.

TinyOS tried to solve these problems by introducing a three-tiered hardware abstraction architecture, comprised by a *Hardware Presentation Layer*, a *Hardware Adaptation Layer* and a *Hardware Interface Layer* [5]. The Hardware Presentation Layer is placed directly over the underlying hardware, and *presents* the hardware to the operating system. Components in the Hardware Presentation Layer are unique for each device they present, but may share a common structure. The Hardware Adaptation Layer groups the hardware-specific components into domain-specific models, such as `Alarm` or `ADC Channel`. The Hardware Adaptation Layer provides the "best" possible abstraction in terms of effective resource usage, but also tries not to hinder application portability. The Hardware Interface Layer uses the adapted components to implement platform-independent abstractions. The TinyOS application developer may choose to use any of the available interface levels, trading off application portability and efficient resource usage.

The *Mantis Operating System* [1], developed by the University of Colorado, aims at making the task of programming a sensor network as close as possible to the one of programming a PC. Thus, Mantis OS uses the classical model of multi-layered operating systems, which includes multi-threading, preemptive scheduling, and a network stack. Hardware is abstracted through a *UNIX-like* API of device drivers, with a monolithic hardware abstraction layer.

While it is true that the classical OS structure used in Mantis OS may lower the learning curve for novice sensor network developers familiar with an *UNIX-like* system, it is uncertain whether this model translates well to such resource-restricted platforms as sensor nodes.

Higher level abstractions often rely on Virtual Machines that abstract the physical hardware into an ideal Virtual Architecture. An underlying translation mechanism ensures correct operation, regardless of specific physical hardware platform details. Thus even highly efficient Virtual Machine implementations may introduce excessive overhead into the system. This is the approach followed by *Maté* [9] a project from the University of California at Berkeley. Maté introduces a high-level interface that allows programs to be replicated throughout the network, reprogramming the nodes in an energy-efficient way. It also provides a safe execution environment, implementing a user/kernel boundary on devices that lack hardware protection mechanisms. In spite of its advantages, Maté suffers from the problems inherent to virtual machines, which are especially critical for highly constrained hardware such as sensor nodes, as practical high level abstractions for Wireless Sensor Networks hardware must make efficient use of the sensor node's low memory, processing and energy capabilities.

## 3 Application Oriented System Design

In this section we present the concepts of *system abstractions* and *hardware mediators* in the context of *Application Oriented System Design* as efficient, high-level, reusable hardware abstraction components for sensor networks. *Application Oriented System Design* (AOSD) [3] was proposed as a multi-paradigm methodology for system software design that makes use of several programming and software engineering techniques that can be combined in order to generate run-time support systems configured and optimized for specific applications.

The EPOS operating system was implemented following AOSD techniques. EPOS aims at allowing programmers to write architecture-independent applications, and, through automated application analysis, building run-time support that complies all the resources that specific application needs, and nothing else. In order to achieve these goals, EPOS relies on the concepts of *Inflated Interfaces*, *System Abstractions*, *Scenario Aspects*, *Configurable Features*, and *Hardware Mediators*. EPOS makes use of *Static Meta programming* and *Aspect-Oriented Programming* techniques to implement software components, thus conferring them a significant advantage over the classic approaches of VMs and HALs. From the definition of the scenario in which the component will be deployed, it is possible to adapt it to perform accordingly without compromising its interface nor aggregating useless code.

### 3.1 Hardware Mediators

*Hardware mediators* are software constructs that mediate the interaction between operating system components, called *system abstractions*, and hardware components. The main idea behind hardware mediators is not building universal hardware abstraction layers and virtual machines, but sustaining the *"interface contract"* between system and machine. Differently from ordinary HALs, hardware mediators do not build a monolithic layer encapsulating the resources available in the hardware platform. Each hardware component is handled via its own mediator, thus granting the portability of abstractions that use it without creating unnecessary dependencies. Hardware mediators are intended to be mostly metaprogrammed and therefore dissolve themselves in the abstractions as soon as the interface contract is met. In other words, a hardware mediator delivers the functionality of the corresponding hardware component through a system-oriented interface.

Hardware mediators are organized in families whose members represent the significant entities in the domain. For instance, a family of CPU mediators would feature members such as ARM, AVR8, and PPC. A part of the UART mediator for the AVR processor is presented in figure 1. This example shows how the system can abstract architecture specific issues such as assembly instructions and register addresses. The AVR_UART default constructor configures the UART with 9600bps 8N1 (there are other constructors to allow the user to configure this device differently). This device is configured through the UART registers. To allow the portability of the UART mediator to different AVR cores this implementation uses in and out methods from de CPU mediator (AVR8), which abstracts the assembly language operations responsible for reading and writing to registers in IO ports into high level C++ methods without any overhead.

```
class AVR8_UART: public UART_Common { // ...
public:
    AVR8_UART() {
        AVR8::out8(UBRR0H,DEFAULT_BR_H);
        AVR8::out8(UBRR0L,DEFAULT_BR_L);
        AVR8::out8(UCSR0C,UCSZ1 | UCSZ0);
        AVR8::out8(UCSR0B,TXEN | RXEN);    }
    ~AVR8_UART(){
        AVR8::out8(UCSR0B,~TXEN & ~RXEN); }
};
```

**Figure 1. The AVR_UART hardware mediator**

### 3.2 System Abstractions and Inflated Interfaces

AOSD relies on *Scenario-independent system abstractions* to implement the operating system components. These components define the system functionalities, and are strongly configurable. System Abstractions are collected from an *Application-Oriented Domain Analysis and Decomposition* process. This analysis process is similar to *object-oriented decomposition*. System abstractions are organized according to the *Family-Based Design* paradigm, and have their *commonalities* and *variabilities* explored through different class hierarchies. An *inflated interface* exports the family as a *super* component, that implements all responsibilities assigned to the family. This component is derived from the interfaces of individual family members, and realized through their implementations. In EPOS, the system framework automatically selects interface realizations, taking into account the target hardware configuration and a cost model for components.

### 3.3 Scenario Aspects and Configurable Features

In AOSD, non-functional aspects and cross-cutting properties are factored out as *scenario aspects* that can be applied to family members as required. For instance, families like UART must often operate in exclusive-access mode. This could be achieved by applying a share-control aspect to the families. *Configurable features*, on the other hand, designate features of mediators that can be switched on and off according to the requirements dictated by abstractions. A configurable feature is not restricted to a flag indicating whether a preexisting hardware feature must be activated or not. Usually, it also incorporates a *Generic Programmed* [10] implementation of the algorithms and data structures that are necessary to implement that feature when the hardware itself does not provide it. An example of configurable feature is the generation of CRC codes in a RF_Transceiver mediator.

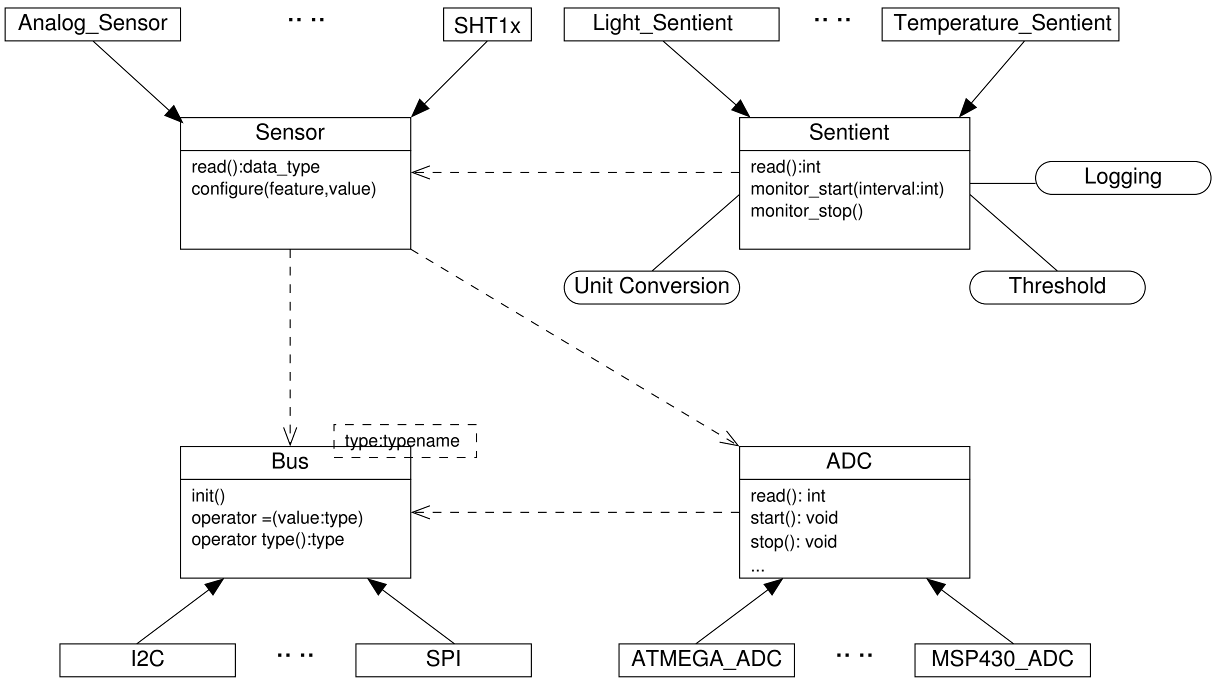

**Figure 2. Set of relevant sensing components.**

## 4 EPOS Sensing Components

In order to preserve the portability of its software components, EPOS relies on hardware mediators. In principle, none of EPOS's abstractions interact directly with the hardware, utilizing the corresponding hardware mediators instead. In this way, a context switch done in the realm of a `Thread` abstraction concerns mainly the decision of which thread should occupy the CPU next, leaving the operation of saving and restoring the CPU's context to the corresponding mediator. This section describes EPOS's sensing subsystem, which provides application programmers with *Sentient* abstractions that use *Sensor* mediators to implement platform-independent sensing components.

### 4.1 Sensors and Sentients

There are several different architectures of sensors in use today, which makes it a hard job to define a common way to access such devices. These devices range from simple, digital integrated circuits to complex, usually analog devices.

Devices in the first category are usually implemented over a serial bus (e.g. I2C, SPI). In this kind of sensors, the data acquisition process is initiated by a specific signal or by read and write operations. Another characteristic of these sensors is that there is a regular time period in which new values are sensed. Examples of this kind of sensors are the Texas TMP family of temperature sensors, most Honeywell Magnetometers, the STM Accelerometers and most A/D converters. On the other end, we have very specific and, generally, analog circuits. Examples of this category of sensors are the Berkley's Mica Sensor Board light and temperature sensors. These sensors share the same analog circuit, making its use and management a complex issue. This circuit has operation and timing constraints, and is composed by two thermistors and one photo resistor, which are managed through 3 GPIO pins, and deliver their results to the same A/D pin. Another example is the Mica Magnetometer. It is implemented by a complex analog and digital circuit and in spite of being configured by a simple I2C potentiometer, it has an analog output and complex semantics and timing characteristics.

Dispite the diversity, it is possible to affirm that sensor operation does follow a *regular pattern*. This pattern is constituted by an *start sensing* command, followed by an optional configuration phase. Once configured, the data acquisition cycle can begin, always respecting the sensor's latency. The sensing routine can then be stopped, or may be kept in a *wait acquisition / get data* loop.

In this context, EPOS defines two entities to encapsulate and, along with other abstractions and mediators, abstract sensor devices: `Sensor` and `Sentient` (figure 2). These entities make use of the `Bus` and `ADC` components to build a comfortable programming interface for sensing application development. The `Bus` mediator family of components implements a uniform way to access peripheral devices. The `ADC` family of analog-to-digital converters mediators also implements an uniform access to such devices. These two basic mediators' families are the key to a modular implementation of the EPOS's sensing subsystem, for they allow the `Sensor` family of mediators implement highly configurable and reusable components. Some of the mediators for the `Sensor` family are also shown in figure 2. Every other implementation just needs to extend the `Sensor` component and make its hardware dependent implementation.

```
/* TinyOS Sensing Application */
configuration SenseToUART {}
implementation {
  components Main, SenseToInt, IntToUART, TimerC, DemoSensorC as Sensor;
  Main.StdControl -> SenseToInt;
  Main.StdControl -> IntToUART;
  SenseToInt.Timer -> TimerC.Timer[unique("Timer")];
  SenseToInt.TimerControl -> TimerC;
  SenseToInt.ADC -> Sensor;
  SenseToInt.ADCControl -> Sensor;
  SenseToInt.IntOutput -> IntToUART;
}
```

```
/* Mantis Sensing Application */
#include <inttypes.h>
#include "led.h"
#include "dev.h"
#include "com.h"
static comBuf send_pkt;
void start (void) {
   send_pkt.size=1;
   while(1) {
     dev_read(DEV_MICA2_TEMP, &send_pkt.data[0],1);
     com_send(IFACE_SERIAL, &send_pkt);
   }
}
```

```
/* EPOS Sensing Application */

#include <sentient.h>
#include <uart.h>

Temperature_Sentient t;
UART u;

int main()
{
  while(1)
    u.send_byte(t.read());
}
```

**Figure 3. Sample sensing applications**

The Sensor family of hardware mediators allows the design of a higher-level, architecture-independent family of abstractions, which can be used by applications without affecting its portability. The Sentient family is comprised by software components that aims in abstracting the sensors finality, and not its implementation (which is considered in the Sensor mediator). Example members of this family are the Temperature_Sentient and the Light_Sentient components. These components implement not only transparent access for sensors, but also some functionalities such as unit conversion, threshold comparison of results, and data logging, which are mapped as configurable features or aspects. Once the application uses a member of the Sentient family, it is up to the EPOS framework composition rules to maintain the sensing sub-system coherence, by granting that, for example, the Temperature_Sentient abstraction will use an available temperature Sensor mediator. This is ensured by the system configuration and generation process, which takes into account the application analysis results and the hardware platform description [13].

## 5 Evaluation

In order to test the expressibility, portability and cost of the EPOS sensing subsystem, we implemented a simple sensing application using three different operating systems: TinyOS, Mantis and EPOS. The application in question implements a loop that constantly reads data from a sensor and redirects the acquired data to a UART. Whenever it was necessary to implement architecture-dependent code, the Berkeley Mica2 mote [8] was considered as the target platform. Figure 3 presents the implementation of this application for the three analyzed operating systems.

The application for TinyOS was written in the NesC programming language, and its implementation consists simply in connecting inputs and outputs of interfaces (ADC and UART, in this particular case). Had it been necessary, the implementation of algorithms could be done in NesC, with *C-like* syntax. The application for Mantis was written in C, including the headers defining Mantis's device access and communication APIs. The application for EPOS was written in C++, importing the sensing abstraction and the UART mediator headers.

In the Mica2 platform, the TinyOS system reflects the hardware design and exports the temperature sensor as an ADC (in the physical platform this sensor is analog and is connected to the micro-controller's ADC). This dependency between hardware and operating system will certainly bring implications to the application portability when, for example, it is ported to a platform in which the temperature sensor is digital and is connected directly to microcontroler IO pins. Even if the application functionality remains the same, it will have to be modified taking into account the details of the hardware platform in which it will be executed.

A similar problem occurs in the implementation for the Mantis operating system. The temperature sensor is read through a device access function, which takes as a parameter the physical device the application wishes to read. Evidently the physical sensor model may vary from platform to platform, and the application will not be portable between different platforms, even when it maintains the same functionality. This problem could be partially solved in the Mantis through a series of `define` statements that would indicate that, for example, in the Mica platform, the `Temperature_Sensor` symbol denotes `DEV_MICA2_TEMP`. Nevertheless, this would still be an inefficient and inelegant solution, as the `dev_read` method aggregates code for the reading of every device available in the platform, even when some of them are not used by the application.

The implementation for the EPOS system does not present any dependency from the target hardware, except for the requirement of an available temperature sensor and a UART, and is perfectly portable between platforms the satisfy this requirement. The selection of hardware mediators is resolved by the system framework, taking into consideration the interfaces used by the application and the target platform specified by the programmer.

All three applications were compiled for the Mica2 as suggested by the respective system user's guide. The footprint of the resulting executable images (application and operating system) is presented in table 2. The application compiled for the EPOS system presented the smallest cost in bytes between the three test systems, for both data memory and code size.

| | Mantis | TinyOS | EPOS |
|---|---|---|---|
| .text (bytes) | 22486 | 9990 | 5522 |
| .data (bytes) | 74 | 16 | 22 |
| .bss (bytes) | 711 | 358 | 152 |

**Table 2. Generated application sizes**

Evidently, an evaluation based simply in program code and data sizes is incomplete, but in resource-limited systems such as sensor nodes, these values are of uttermost importance. Future evaluations of this work will include performance and energy-consumption measurements.

## 6 Conclusions

We discussed the problem of heterogeneity in sensor networks hardware, and presented a novel technique to handle application portability in these systems. We used the concepts of *hardware mediators* and *system abstractions* to model and implement the EPOS sensing subsystem, which allows sensing applications to be ported between different sensor networks hardware platforms. A sample sensing application was implemented for the EPOS operating system, as well for other available operating systems for sensor networks. The EPOS system presented the smallest size int terms of code and data size. These results position EPOS as a very viable alternative for operating systems for sensing devices.

## References

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications*, pages 50 – 59, 2003.

[2] Atmel Corporation. *AT86RF401 Datasheet*. San Jose, CA, Dec. 2003.

[3] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation*, San Diego, CA, June 2003.

[5] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, and A. W. D. Culler. Flexible hardware abstraction for wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05)*, 2005.

[6] J. Hill. *System Architecture for Wireless Sensor Networks*. PhD thesis, University of California, Berkeley, 2003.

[7] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, 2004.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, Cambridge, Massachusetts, United States, 2000.

[9] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, Oct. 2002.

[10] D. R. Musser and A. A. Stepanov. Generic programming. In *Proceedings of the First International Joint' Conference of ISSAC and AAECC*, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.

[11] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, Los Angeles, California, Apr. 2005.

[12] J. Rabaey, J. Ammer, T. Karalar, S. Li, B. Otis, M. Sheets, and T. Tuan. Picoradios for wireless sensor networks: The next challenge in ultra-low-power design. In *Proceedings of the International Solid-State Circuits Conference*, San Francisco, CA, Feb. 2002.

[13] G. F. Tondello and A. A. Fröhlich. On the Automatic Configuration of Application-Oriented Operating Systems. In *3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, Jan. 2005.