
Luiz F. M. Vieira, Breno A. D. Vitorino, Marcos A. M. Vieira, Diógenes C. Silva, Antônio O. Fernandes
Antonio Alfredo F. Loureiro

Computer Science Department and Electrical Engineering Department
Universidade Federal de Minas Gerais
Avenida Antônio Carlos, 6627 – Pampulha
Belo Horizonte – MG – Brazil
{lfvieira, vitorino, mmvieira, otavio, loureiro}@dcc.ufmg.br, diogenes@eee.ufmg.br

Abstract

Through a new computing paradigm, the wireless sensor networks revolutionize the environment monitoring, enabling applications in a wide range of knowledge areas. However, the process of creating these applications for these networks is still not simple, depending on the target operating system and hardware. In order to fill this gap, we developed a new programming model and a framework able to generate the application source code for a user-chosen computing platform, based on a modular specification. This novel approach is easy to program and allows the generated code to be efficient, as efficiency is a paramount requirement due to severe restrictions on WSN. Two case studies are described demonstrating the potential of the framework to help software developers.

1 Introduction

Recent advances in hardware miniaturization and wireless communication created a new paradigm on environment monitoring [5]. Small, low-cost devices denominated sensor nodes and empowered with communication, sensing and processing unit can be deployed in inhospitable places. Collaborating through wireless means, these devices can measure environment characteristics such as pressure, temperature or luminosity [8]. These data can be processed internally in the network and later sent to an access point which has a global view of the network and that can be used for a different number of applications. Some of these applications include fire detection, precision agriculture, and target tracking. The collection of these wireless devices composes a wireless sensor network (WSN).

While programming the nodes of a WSN, the software designer has to deal with concurrent computation. For example, an application has to be able to react to routing messages while applying a filter to the collected data before sent over the radio. Concurrency issues, such as scheduling and data racing are handled directly by the software designer, using abstractions provided by a C-like language and, optionally, by an operating system. These limitations lead to hard-to-understand, error-prone programs, as these programs aggregate more functionality.

The software developer has also to deal with hardware specific components. For example, to program an application to send a message, she needs to understand how the radio works. This could lead to longer software development time or even, incorrect applications. Furthermore, no reuse code is provided.

To overcome these issues, we propose WISDOM, a visual development framework that is capable of generating native code for a given computing platform from a modular, platform-independent specification. We state a computing platform as the combination of an operating system and the hardware where it runs. In an application modeled with WISDOM, each module abstracts an application task, and links between modules define the execution and data flow. Pre-built modules define methods for accessing the hardware devices such as sensors and radio. Users can specify their own modules to determine the application’s logic.

The main objective of this new framework is to speed up the application development time for WSN and to allow a one-description many-platform development. The programming model only assumes the understanding of ANSI-C language. No C libraries have to be used or learned because it already abstracts the hardware access. The modular programming adopted in WISDOM is conceptually clean and enables code reuse, which is responsible for the decrease of time in implementation and testing. The provided framework also enables the visualization of the modules and its dependencies in a given application.
The programming model adopted by WISDOM includes an intuitive abstraction to concurrent processing. No data is explicitly shared between modules. When a module determines that its data is ready, it posts the data to the other modules. After the post, this module can process new incoming data. This behavior enables high concurrency in an event-driven based execution.

A major advantage of WISDOM is its programming model, which is independent of the computing platform that will be executing the application. Therefore, an application specified with WISDOM is portable to any platform which provides the functionalities required by the application.

2 Wireless Sensor Networks

In this section we describe WSNs and current WSN platforms. WSN are composed by sensor nodes, which can be of different platforms, forming a heterogeneous WSN. The current tendency is to integrate solutions. Programming each sensor node of a different platform would require the development of an application for each of them, without any code reuse. In order to avoid that, WISDOM defines its programming model which is described in section 3.

Nowadays, WSN is a rich field of research and, as the MEMS technology evolves, more are being developed. Although these platforms have their particularities, they share some basic concepts that we based for constructing a platform-independent programming model. Here we describe four known platforms.

2.1 TinyOS and Mica

This computing platform is developed at UC Berkeley, and is currently the most popular. Its sensor node, the Mica mote, is comprised of an ATMEL ATMEGA processor, CC1000 radio transceiver, a sensor board, external EEPROM memory, and its battery [7]. The sensor board includes light and temperature sensors, two-axis accelerometers, and a magnetometer. TinyOS was the first operating system designed to run in this hardware.

TinyOS [6] contains an event-driven scheduler. This scheduler introduces the concepts of events and tasks. Events are software abstractions for hardware events such as sensor sampling and message receiving. The hardware events are the response of non-blocking requests to hardware called commands. Events have the highest priority, execute to completion, preempt and schedule tasks. Tasks contain the main computation of an application and are dispatched by the scheduler. Tasks also execute to completion and schedule other tasks.

TinyOS applications are written in nesC [4], an extension to C language. This language includes the syntax for expressing tasks, events, and components. Components aggregate similar functionalities and enable modular programming. The framework uses the generate programs in nesC when this platform is selected.

2.2 Mantis OS and Nymph

This computing platform is developed by the Mantis group at University of Colorado. The Mantis OS (Mantis Operating System) was first designed to run on Nymph, the Mantis group sensor node. Nymph [11] is inspired on Mica architecture, and has the same processor and radio modules. The only difference is that its sensor and programming interface are at the processor/radio board.

The Mantis OS [11] provides a subset of POSIX threads. Threads are scheduled to execution when a timer interrupt triggers or at system calls or semaphore operations. Hardware access is done by blocking calls. It is clearly not an event-driven scheduler, but it is possible to simulate one by scheduling one thread per hardware event.

The Mantis OS exports a C language API that can be easily incorporated by WISDOM to implement programs using this computing platform.

2.3 PeerOS and EYES

The EYES european research project developed their own computing platform: the PeerOS operating system and the EYES sensor node. The EYES sensor node [13] uses the MSP430 microcontroller as its main processor and the TR1000 radio transceiver. It also contains an external memory, battery, and sensor board, similar to the Mica mote.

PeerOS has a preemptive task scheduler based on the EDFI algorithm [10]. This algorithm guarantees some hard real-time characteristics with little context switch overhead. We believe that this computing platform can be integrated to WISDOM if we take an approach similar to the Mantis one.

PeerOS’s functions are written in C, and can be easily supported by WISDOM framework.

2.4 Yatos and BEAN

Yatos operating system and BEAN sensor node were developed by SensorNet group. BEAN [14] has a MSP430F149 processor, the CC1000 radio transceiver and the connector for the sensor board. This sensor node also enables to measure the current at each component.

Yatos [14] is an event-driven operating system. It has the concepts of events and tasks, similar to TinyOS. Application developers can declare which hardware events they are interest in and associate tasks to each of these events. Whenever one of these events occurs, the associated tasks are scheduled in a priority queue. When there are no tasks in the scheduler, BEAN is put in low-power mode.

Yatos’s API is written in C, so ANSI-C language can be used to write applications that use this operating system.

3 WISDOM Programming Model

In this section we present the concepts involved in WISDOM Programming Model. This model was designed to describe the application at a given sensor node of the network, and not the application at the whole network.
perspective. For example, two nodes can run different applications that collaborate to reach the same objective. A user can use this model to define the communication, processing, sensing and the node’s behavior as part of the network. From this definition, the WISDOM framework can generate native source code for any computing platform it supports.

Before describing the concepts, we first state some model’s assumptions that guarantee its platform-independence. Those assumptions are not restrictive: all platforms described in section 2 present the following:

- The operating system exports its functionalities through C routines. The application is expressed using the ANSI-C language. For this reason, we suppose that it is possible to integrate the application with the operating system facilities to build a functional application.
- The operating system has an event-driven scheduler. All communication with the hardware modules is supposed to be done in a non-blocking manner. Therefore, a hardware interrupt triggers whenever some device has valid data to be handled. An event-driven scheduler only executes some computation in response to hardware interrupts. The code that executes on these responses has to be resolved quickly, since it ignore other interrupts. The scheduler has to allow the execution of longer computations at a lower priority.

Any application in WISDOM is composed of modules. A module is an abstraction for a set of correlated functions. For example, there are modules for accessing a specific sensor, to send and receive messages over the radio, or to filter some data set.

There are two kinds of modules: system modules and user modules. System modules are pre-built modules which abstract the hardware access at the sensor node. They initiate any computation in the application, stimulated by the data received from the environment. Their fields and methods cannot be modified by the user; instead, their method headers are well-defined and their implementation is dependent of the underlying platform. The method header definition includes the careful description of its inputs and outputs, so that the user can exactly know the behavior of these modules. User modules, in the other way, can be freely specified by the user. They abstract other application’s functionalities not covered by the system modules, such as data filtering and message handling.

All modules are composed of fields and methods. Methods are the procedures available to the other modules, and fields are the internal variables, accessible only by its module’s methods. All the fields and methods are described in ANSI-C language, which is supported by most platforms. There are two important types of methods: signal methods and receiver methods.

Each signal method of a module is associated to an interest event for the other modules. For example, this event can a message reception or a sensor reading (for system modules), or the conclusion of some computation or an occurrence of an internal exception (for user modules). Hardware events, such as the ones handled by system modules, are supposed to set hardware interrupts, which are wrapped by the signal methods of these modules. On the other hand, user modules should define algorithmically when their signal methods should be called. As they signal an event, only the headers of these types of methods have to be defined. To effectively handle the events advertised by these methods, receiver methods should be created. The receiver methods are capable of sending messages over the radio, blinking the leds, filtering incoming data, or handle some exception. A receiver method is invoked each time its associated signal methods are executed.

To associate signal methods to receiver ones, one should create a connection between them. The connections are defined statically, during the application development. A connection between signal and receiver methods is possible only when the type and the order of declaration of their parameters match. The cardinality of a connection is many-to-many, such that one signal method can execute many receiver methods in an arbitrary order, and one receiver method can be dispatched by multiple signal methods.

The set of chosen modules and their connections define the execution and data flow of an application. The model works with an event-driven execution, so all computation is started by hardware interrupts, which are signal methods of system modules of our model. These signal methods always carry some data with them, so that the associated receiver methods can handle these data. Thus, the execution flow defines where data will be processed and transferred. From these system modules, the execution flow continues through zero or more user modules, and eventually reaches other system module. This last system module is responsible for transmitting the processing data out of the node, through wireless means (to other nodes) or cabled connections (for gateway nodes).

4 WISDOM Framework

The WISDOM framework is a tool for developing programs for nodes of a WSN in accordance with the programming model described in the last section. It provides a graphical user interface to aid the developers. Through this interface, it is possible to insert modules in an application, define signal and receiver methods, establish connections between them, and finally generate code for a computing platform. Currently WISDOM supports the following platforms: TinyOS/Mica Motes and Yatos/BEAN. It is implemented in Java, and it is open-source.

This framework’s main features are:
Visual programming support. The visualization of the modules and their connections permits the user to have a better view of its application’s execution and data flow, which implies in quicker developing and testing. We also hope that this feature can facilitate the programming by non-specialists, hiding aspects like creation and managing of concurrent tasks and learning to use a new language and/or an operating system.

Multiplatform code generation. The WISDOM programming model does not assume any specific computing platform. Thus, from the specification of an application using this model, the WISDOM framework can analyze this specification and generate code for a given supported platform (figure 1). This feature enables code portability, which is important for heterogeneous WSN.

Extensibility. The WISDOM framework was designed to be extended to generate code for more platforms than it currently supports. There is already a set of classes that provides the basic functionality for adding new platforms.

In the later subsections we detail the framework’s architecture and characteristics.

4.1 Software Architecture

WISDOM is implemented in four layers: view, control, core, and platform. These modules and their relationships are shown in figure 2.

The view layer is responsible for WISDOM interface. It contains all the screens, the module and connection graphical representation, and the management of these graphical elements on the screen. A selected module can be dragged on the screen for better visualization of the application.

The control layer links the view layer with the core layer. It means that this layer selects the proper core classes for each user’s action at the interface. This layer simply decouples the interface from the framework’s business logic.

The core layer actually implements the programming module. It has the classes that implement system and user modules, signal and receiver methods, and connections. The application class contains the collection of modules and connections that form an application. This class’s state can be saved or loaded from a file for user’s convenience. This layer also contains the basic logic for code generation, selecting the proper platform classes for this process.

The platform layer contains the specific code for each computing platform supported by WISDOM. There are classes containing the implementation of each system module available in the framework. It also contains the logic to generate the proper system calls to handle a connection of the programming module.

4.2 Framework description and usage

The module in this tool is represented by a rectangle with three fields: the identification field, which contains its name and an icon which represents its functionality; the signal methods field, which shows the current set of signal methods of this module; the receiver methods field, which contains all the receiver methods of this module.

The tool enables the inserting of both system and user modules. The collection of available system modules is shown in a combo box for the user. To create new system modules, it is necessary to include new classes in this tool to represent its module’s definition and to determine the code generation rules for each platform supported by WISDOM. There is also a repository of some user modules that are commonly used in application. A new user module can be easily created by filling its name, description, and signal and receiver methods. This information
can be modified later, at the user’s needs.

The connection between modules is presented as an arrow from the signal method of one module to the receiver method of another module. The connections are created by selecting the source module, its signal method, the destination module, and its receiver method. The connection is then inserted at the list of connections of an application, which can be seen or modified at one screen (figure 3). After confirming the modifications at this screen, the application is updated with the new arrow (or with the removal of arrows, if some of the connections were removed from the list).

The receiver method’s user code can always be viewed and modified at clicking on any receiver method in the module’s graphical representation. The code has to be written in ANSI-C. As the model does not suppose that the underlying operating system supports memory management, dynamic memory allocation is not possible. If the user needs to maintain data across executions of the receiver method, she can declare variables outside the method’s scope. This variable is global but is not accessible by other methods, thus no shared access is possible.

After selecting the user and system modules, making the connections between them and defining the receiver method’s code for any new user modules, the tool is ready to generate code. The user chooses the target platform, and WISDOM translates the modules and its connections to system calls of the operating system. Multiple files are then created after the code generation. These files are ready to be compiled with the tools provided by the target platform. The executable code can be stored at the node’s memory through the JTAG device.

4.3 Code generation issues

The translation of the application specification to the source code for a target platform involves four basic steps: initialization code generation, system modules code generation, connection code generation, and code precedence analysis.

The initialization code is simply some set of calls that have to be made before the application runs. These settings include the hardware initialization, such as clock frequency and enabling or disabling some hardware modules and pins.

Another set of routines are needed to access the hardware modules. These routines are generated according to the system modules used in the application. For example, the Radio system module offers methods send and receive, which are then filled with the system calls related to the link layer that sends and receives messages over the radio. Asynchronous tasks, like message receiving, and analog conversions, such as sensor sampling, are all done in a non-blocking manner. It means that the task will be scheduled to the hardware module and the latter is responsible to set a hardware interrupt when the task is complete.

Next there is the connection code generation. WISDOM reads each connection and dispatches a task for each receiver method that is associated to one signal method. The dispatch is enabled only if it is not already present at the scheduler queue. If this is not the case, the event will be simply ignored by the receiver method. In fact, this is a message passing scheme with a queue of size one. This strategy does not allocate more memory than it was defined by the user, but he has to be aware of the node’s sampling rate to avoid the loss of many events. It is also generic enough to be implemented in any event-driven scheduler. Other strategies can also be used for better performance and/or memory usage.

Finally, the code is written to the files respecting the precedence order of the calls. In C language, it is not possible to call a subroutine if it was not previously declared in the code. This precedence is achieved by first writing the signal methods of the system modules and proceeding through the connections between signal and receiver methods until there is no connected signal method at a given module.

5 Results

In this section we show two applications generated with WISDOM. One application samples the light sensor and shows the sample’s values on the LEDs. The other collects light samplings and sends them over the network. Its routing algorithm implements data fusion in the intermediate nodes. It is important to remark that it is not our objective to present the best solutions for the applications proposed, but to show the potential of our framework to make the node programming easy and to demonstrate our programming model with two case studies.

Both applications were tested on Mica2 motes running TinyOS. We also generated code for Yatos/BEAN, according to the available APIs and test it via simulating tools.

5.1 Case Study: Sensor Reading

The objective of this application is to show at the node’s LEDs the binary value of the light samples over some user-defined threshold. Only the most significant bits are shown on the LEDs, as many as available on the node. For
example, in the Mica Motes there are three leds available, and the most significant bit is shown on the red led. With a simple application, we show every step needed to compose a program in WISDOM.

First, we choose the Sensor_Light and Leds module. They offer the hardware resources we need to accomplish our goal. We also created a Threshold module which will light the leds only when the light readings overcome the threshold. This user module has one receiver method, getReading, which gets the sensor readings and checks its value for the threshold and one signal method, thresholdAchieved, that is invoked by getReading whenever the threshold is exceeded. These three modules are all we need to compose this application.

With all the modules we need, it is now a matter of connecting them. Connecting the signal method of Sensor_Light with getReading in Threshold module means that all reading samples will be handled by this receiver method. Connecting thresholdAchieved from Threshold with the receiver method of Leds will transfer the value above the threshold to the leds conveniently.

After these steps, the application is conveniently modeled. WISDOM can generate code for the chosen platform, which has to be compiled and loaded on the node. The final result is shown in figure 4. The WISDOM generated code for TinyOS and Yatos is shown in listings 1 and 2.

Listing 1. Sensor Reading code for TinyOS.

```c
module SensorLedsM {
    provides {
        interface StdControl;
    }
    uses {
        interface Leds;
        interface Timer as TempTimer;
        interface ADC as TempADC;
        interface StdControl as TempADCControl;
    }
}
implementation {
    command result_t StdControl_init() {
        ...
    }
    command result_t StdControl_start() {
        ...
    }
    command result_t StdControl_stop() {
        ...
    }
    ...
    task _Leds_light() { ... }
    void thresholdAchieved(int nivel) { ... }
    task void getReading() { ... }
    void _Temp_getData(int _Temp_data) { ... }
    event result_t TempTimer_fired() { ... }
    async event result_t TempADC_dataReady(uint16_t data) { ... }
}
```

Listing 2. Sensor Reading code for Yatos.

```c
#include "SensorLeds.h"
#include "SO.h"
...
void _Leds_light(evento_t ev) { ... }
void thresholdAchieved(int nivel) { ... }
void getReading(evento_t ev) { ... }
void _Temp_getData(int _Temp_data) { ... }
void tempTask_Sensor_Temp(evento_t ev) { ... }
void sensorTask_Sensor_Temp(evento_t ev) { ... }
void main() {
    Microcontrolador_IniciaHardware();
    Tarefa_Declarations(&tempoTask_Sensor_Temp, 128, getReading);
}
```

5.2 Case Study: Data Fusion

In this case study, we want to present how WISDOM deals with intercommunication. In this application, nodes have different holes in the network, which is a common scenario for these networks.

We have three different holes: collectors, gatherers and gateway. The collector nodes just sample their lights sensors and send this data to their gatherer through the radio. Each collector has an associated gatherer, and this assignment is done in development time. A node is a gatherer if this address modulus GROUP_SIZE is zero. GROUP_SIZE is a constant that assigns how many collectors will exist for each gatherer. The gatherer hole is to receive the samples from its collectors and fuse them, transmitting to the gateway only the mean of the last ten samples. The gateway node will then receive messages from the various gatherers and send them through a serial port of a common PC. The computer reads the serial and displays all this data on a graph to the user. The described topology is seen in figure 5.

This entire scenario will be described in one application, where the nodes are always testing their address to check their holes in the network. This could be also described in three applications, one for each hole, but we preferred but the former approach because it permits us to test the application first in a simulator like TOSSIM. TOSSIM is able to simulate a WSN where all its nodes are running the same program.
Figure 5. Data fusion network topology. Notice that in this example GROUP_SIZE is 5.

First, we selected the necessary modules for this application. We used the Radio, Sensor_Light, and IOPorts system modules. Radio is responsible for providing the send and receive methods for messages, Sensor_Light for gathering light samples, and IOPorts for transporting messages over the serial port to the PC. We also created some user modules, MeanFusion and Base. The latter is responsible for checking the forward the messages to the serial if the node is a gateway, and the former contains the routing algorithm and the data fusion rules.

The connections between modules are established as follows. Sensor_Light’s is connected to a receiver method in MeanFusion that defines the collector behavior. In the same way, Radio’s signal method receive is connected to another receiver method in MeanFusion that is responsible for the gatherer algorithm. Both receiver methods of MeanFusion call its signal method to indicate that the message is ready to be sent over the network. Thus, this signal method is connected to the send method in Radio, which is of type receiver. The Base module determines the gateway logic, and it is connected to both Radio and IOPorts.

This application in WISDOM framework is shown in figure 6. Its source code for TinyOS and Yatos, generated by WISDOM, is shown in listings 3 and 4.

6 Related Work

The WISDOM programming model was designed using popular design choices for WSN, such as event-driven applications [11] and modular programs [4]. The concept of signal and receiver methods of a module is analogous to inputs and outputs of a module, but their visualization in a framework was inspired on QT [12]. It was designed to be simple, yet complete for the sensor network applications’ needs. We believe that this approach has a smoother learning curve than the other standardized languages such as IEC61499 [3].

We investigated the platforms that WISDOM would apply. They include the currently supported TinyOS/Mica
Motes [6] from Berkeley, and Yatos/BEAN platform [14]. Other platforms like Mantis [1] and PeerOS [10] also have the WISDOM prerequisites: C language support and event-driven scheduling. We hope that these platforms can also take advantage of WISDOM framework for quicker application developing.

As far as we know, this is the first visual programming framework for WSN that generates code for multiple platforms. But there are proposed programming models for WSN, such as TinyGALS [2]. Its objective is to handle common concurrency problems in applications written in TinyOS. It offers a language that allows the separation of synchronous and asynchronous communication. The synchronous part is written in an imperative language similar to C, while the asynchronous communication is specified as connections between components. A compiler translates the connections to lower-level structures such as message passing through queues and guarded variables.

Although TinyGALS language has many similarities with our programming model, our work extended the TinyGALS concepts. The TinyGALS was constructed over nesC and TinyOS concepts. We applied a programming model that is platform-independent, so that the benefits of modular programming can be extended to other upcoming platforms. The visual framework and the adoption the ANSI-C language makes us believe that our approach will reduce the learning curve for novice programmers for WSN.

7 Conclusion

In this work we presented WISDOM, a framework and a programming model for WSN nodes. The user can rapidly assemble a program through the WISDOM framework. The user can select which modules his application needs and connect them to define the application’s execution flows. A graphical interface enables the visualization and modification of these modules and their connections in an intuitive manner. We associated a new platform-independent programming model to the WISDOM framework.

We provided some example applications modeled in WISDOM. The visual environment offered by the framework allows the better comprehension on how the application will work and how its modules interact with each other. All examples were tested on the sensor nodes, validating the code generated by WISDOM.

We envision for future directions of our work the creation of a user module library with useful algorithms for different applications. Mapping the executing code in our programming model, for debugging purposes, is another research topic in our work. We also propose that WISDOM framework can be extended in order to verify program properties modeling the modules and their connections as a Petri Net.

References