A UML 2-based Hardware-Software Co-Design Framework for Body Sensor Network Applications

Zhenxin Sun*, Chi-Tsai Yeh†‡, Weng-Fai Wong*
*Department of Computer Science, National University of Singapore, Singapore 117417
Email: sunzhenx@comp.nus.edu.sg;wongwf@comp.nus.edu.sg
†Department of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan, R.O.C.
‡Department of Information Management Shih Chien University, Kaohsiung, Taiwan, R.O.C.

***Abstract*—This paper proposes a unified framework for the hardware/software codesign of body sensor network applications that aims to enhance both modularity and reusability. The proposed framework consists of a Unified Modeling Language (UML) 2 profile for TinyOS applications and a corresponding simulator. The UML profile allows for the description of the low-level details of the hardware simulator, thereby providing a higher level of abstraction for application developers to visually design, document and maintain their systems that consist of both hardware and software components. With the aid of a predefined component repository, minimum TinyOS knowledge is needed to construct a body sensor network system. A novel feature of our framework is that we have modeled not only software applications but the simulator platform in UML. A new instance of the simulator can be automatically generated whenever hardware changes are made. Key design issues, such as timing and energy consumption can be tested by simulating the generated software implementation on the automatically customized simulator. The framework ensures a separation of software and hardware development while maintaining a close connection between them. This paper describes the concepts and implementation of the proposed framework, and presents how the framework is used in the development of nesC-TinyOS based body sensor network applications. Two actual case studies are used to show how the proposed framework can quickly and automatically adapt the software implementation to efficiently accommodate hardware changes.**

I. Introduction

Body sensor networks (BSNs) are wireless sensor networks (WSNs) that are widely used in health care and biomonitoring systems [1]. These systems continuously monitor human vital signs, detect critical conditions, and alert care-givers or medical professionals accordingly. In such systems, hardware and software often need to be co-designed and co-optimized. The unique features of BSN bring the following three challenges to their designers.

The first challenge is that the level of abstraction is low in the current practice of BSN application development. A number of BSN applications are currently implemented in nesC [2], a dialect of the C language, and deployed on the TinyOS [3], which provides low-level libraries for basic functionalities such as sensor reading and node-to-node communication. While nesC and TinyOS do hide some amount of hardware-level details, the low level of abstraction does not help developers to rapidly prototype their applications because reusability in the design is low. The second challenge is that any emergency or anomaly in the vital signs of a patient has to be processed within hard real time deadline constraints. Finally, because the BSN components are wearable and mobile devices, there is a need to maximize battery life.

This paper proposes a new model-driven development framework that is intended to manage the complexity of BSN application development by addressing the above issues. The framework consists of (1) a Unified Modeling Language (UML) profile for TinyOS to capture the specifications of BSN applications, and (2) a user customized simulation environment to perform validation and verification. In our framework, both design and validation are unified under a high level specification language, namely UML, thus yielding a complete and holistic flow that supports the design, implementation, simulation, and refinement cycle of rapid prototyping. Refinement and tuning will also benefit from the formalized reuse model. We shall present several actual case studies to show how TinyOS implementations were produced from specifications and how refinement was done effectively with the aid of simulation.

II. Related Works

Several UML profiles have been proposed to address various design issues [4]–[7]. The main novelty of our paper is a UML design flow specifically targeted at BSN applications. Unlike general embedded systems or WSNs, BSN applications have centralized control, and a fewer number of more sophisticated motes.

Several simulators, such as OMNET++ [8], J-Sim [9], Em* [10], have been built to analyze runtime behaviors of TinyOS applications. However, none of these support the cycle-accurate simulation of BSN node devices. One simulator that has modeled both the processor and the peripherals of sensor nodes for cycle accurate simulation is the Avrora simulator [11]. Avrora executes standalone applications and does not integrate into a high-level design flow like ours.

Gratis [12] provides a graphical design environment to create and generate TinyOS implementation. The main aim of Gratis is to model the structure of the TinyOS application. It therefore does not model the behavior of the application. Our framework captures both the structure as well as behavioral specifications at the UML level and distinguishes itself from others described above in enabling the reuse of both the design and simulation components. We argue that this will significantly reduce design and validation cost.

III. UML 2-based Framework for TinyOS-based BSN Applications

A. UML Profile for TinyOS

Table I
UML profile for nesC-TinyOS.

Element	Keyword/Concept	Purpose
Classes	`<<SystemDefined>>`	Predefined nesC module
	`<<UserDefined>>`	User-defined module
Operations	`<<command>>`	Inter-component communication
	`<<task>>`	for intra-component concurrency
Tag types	`sync`	Synchronous commands/tasks
	`async`	Asynchronous commands/tasks
Interfaces	Required	Requests that are made from the class to its environment. A required interface is denoted by a socket notation, '('.
	Provided	Requests that are made from the environment to the class via a port. A provided interface is denoted by a lollipop notation, '◯'.

The basic components in TinyOS programs are modules. There are two kinds of modules in a TinyOS application: `predefined modules` from the nesC library and `user defined modules`. Each module is modeled as a UML class. The stereotype `<<SystemDefined>>` is used to identify predefined modules supported by the nesC library. Similarly, the stereotype `<<UserDefined>>` is given to UML classes that model user-defined modules. Commands and tasks are modeled as operations, and they are differentiated using the stereotype `<<command>>` and `<<task>>`, respectively. Tag types `sync` or `async`, are introduced to indicate whether a command is synchronous or asynchronous. The module interfaces are modeled as UML ports attached to the classes. A UML port, which is denoted by a square notation □, defines a distinct point of interaction between a class and the environment. The UML port takes the name of the nesC interfaces it specifies. Table I summarizes the proposed UML notations for TinyOS.

B. Modeling Behaviors

In TinyOS, all the components are running concurrently and execution is preemptive. This is hard to model in a single *state machine diagram*. TinyOS is a event-driven system, and all the events have to be sent through the corresponding interfaces. We use a combination of *activity diagrams* and local *state machine diagram* to model system behaviors. Each *structure diagram* comes with a corresponding *activity diagram* to capture the interactions between the components. Using *activity diagrams*, the user will have a clearer picture of the interactions in the system. To the *activity diagram*, the designer can add the trigger actions to the corresponding *state machine diagram*. Each action specified on the association indicates a trigger event. The trigger actions will be specified using an *interface state machine*.

C. Code generator and implementation for BSN Application

Both the basic modules and the application models are used for code generation. Each class with the stereotype `<<UserDefined>>` is transformed into two parts: a nesC module and a configuration. The attributes and operations defined in the model will be translated into variables, and methods or tasks. Links will be translated as "wires". The operations of classes that implement interfaces will be extracted and added to its aggregate module. The implementation of an event is derived from the corresponding *state machine diagram*. Specifically, these are the actions embedded under the trigger action.

A code generator called `UML2nesC` has been implemented to produce the nesC implementations. The generators takes UML models as input, extracts structural and behavioral specifications and constructs an abstract syntax model for nesC. The intermediate models are merged with our defined nesC templates to produce the implementation code in nesC. The interfaces were generated from the links of structural model. Each event specified in the *activity diagram* maps to an event trigger action, while the detailed implementation was generated from the *state machine diagram*. The connections of the interfaces translate to "wires" in the nesC code. For each link in the *structure diagram*, a corresponding interface connection was created.

D. Code generator and implementation for BSN Simulator

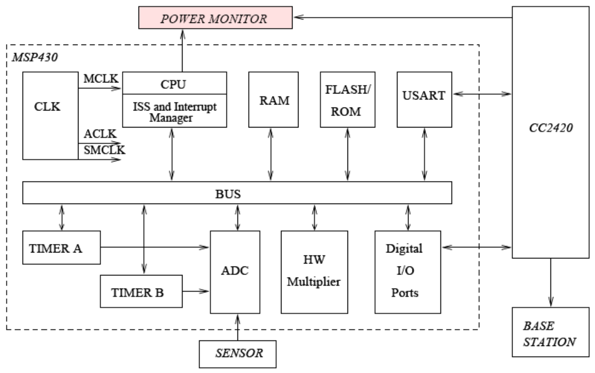


Figure 1. Structure of BSN simulator.

Our BSN simulator is an extension from this work [13]. The simulator is written in SystemC and can achieve accurate timing and energy estimates. Figure 1 shows the structure of our simulator. We extend pervious work by capturing the simulator in SystemC UML profile. By doing so, the simulator portion of overall design framework is unified in our UML-based framework, and the simulator can be easily customized and managed through high level modifications. With exchangeable UML components, we are able to easily change or replace the constructed hardware component in the model. Thus it allows for re-configuration of components, or the "plug-and-play" of new components, making it ideal as a validation platform for the pre-integration stage.

E. Design Methodology

The TinyOS repository serves a key role in the framework. It contains a set of UML classes formalized using our proposed UML profile. To design a new application, designer can either start with a similar application from the repository or design one from scratch using the modules in the repository as building blocks. The nesC library provides essential components such as timing, storage, analog to digital converter (ADC), and others. All system modules will be modeled as UML classes with stereotype `<<SystemDefined>>` and maintained in the repository together with user-defined modules. Figure 2 shows design flow for BSN applications using our proposed automatic UML2-based framework. Designer will start to build new application for user requirement and then partition that into hardware and software. If the new TinyOS modules are required for software design, the designer will need to construct them using proposed UML profile for TinyOS. The newly defined modules are then added to the TinyOS repository, and the similar flow is processed in hardware design. Our UML parsing engines convert these UML text files into hardware simulator (SystemC) and application software (nesC) implementation separately. An application software is built by using *structure diagrams* and *state machine diagrams*. The generated nesC code can be used for deployment in the simulator. The designers can obtain the simulation results rapidly using our proposed framework, and then proceed to analyze the results in order to refine the hardware/software design.

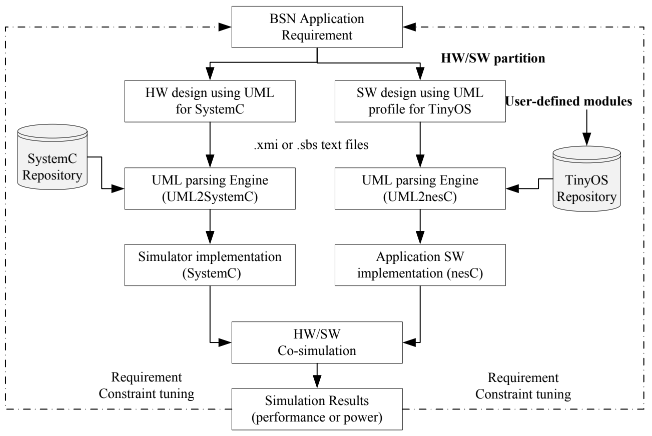

BSN Application Requirement
HW/SW partition
HW design using UML for SystemC
SW design using UML profile for TinyOS
User-defined modules
.xmi or .sbs text files
SystemC Repository
UML parsing Engine (UML2SystemC)
UML parsing Engine (UML2nesC)
TinyOS Repository
Simulator implementation (SystemC)
Application SW implementation (nesC)
HW/SW Co-simulation
Requirement Constraint tuning
Simulation Results (performance or power)
Requirement Constraint tuning

Figure 2. Our proposed design flow for BSN application.

IV. Case Studies

A. Wheeze Detection

Our first case study is an audio based health monitoring system that consists of three components: a microphone array, preprocessors for the audio signal, a mote, and a PDA. The application is designed to run on a TinyOS mote that periodically samples data from the data collection module. Features are extracted from the data and compared with certain pre-determined thresholds. If a sequence of audio signal is classified as a wheeze, it will be sent to PDA for storage. We start the design by adding in a number of predefined nesC components: `SerialActiveMessageC`, `ActiveMessageC`, `TimerMilliC`, `mainC`, `LedsC`, and `MSP430ADC12P`. A control module, `WheezeDetection`, is then added in and marked as `<<UserDefined>>`. Buffers and coefficients are entered as class properties and other utility functions are added as class methods. UML ports are added to the `WheezeDetection` module, and are connected to corresponding UML ports of predefined modules. The *structure diagram* of wheeze detection system is shown in Figure 3.

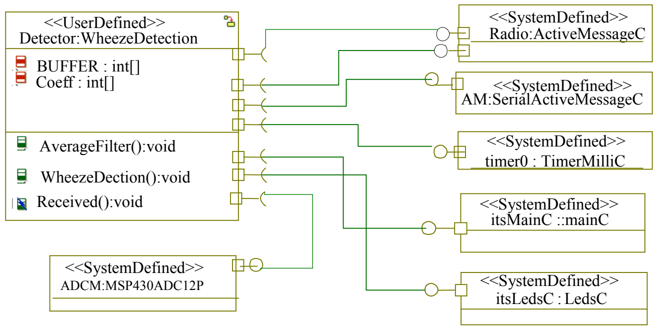


Figure 3. Structure diagram of wheeze detection application.

The *activity diagram* in Figure 4 is used to capture the interaction between the modules. We need to add all the object instances into *activity diagram*. Lines and message calls are added if two modules have interactions. For example, the event `Boot()` of interface `Boot` will trigger the initialization of `WheezeDetection` module. After the initilization, `WheezeDetection` will start to process the data collected from other modules. Each `DataReady()` event from `MSP430ADC12P` will trigger `WheezeDetection` to store the incoming data in a buffer, and when `TimerMilliC` fires every 32ms by the `fired()` event, `WheezeDetection` will process the stored data within next 32ms and send a message through `ActiveMessageC`, and then wait for the `SendDone()` event to indicate the message is sent so that it can continue in next processing cycle.

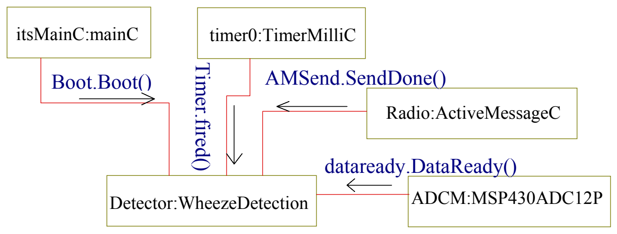


Figure 4. Activity diagram of wheeze detection application.

B. ECG and SpO2 Monitor

The second case study is a MEMSWear bio-monitoring application, namely the SpO2nECG application taken from [14]. In this application, a single sensor mote collects data from the attached sensor at a sampling rate of 250Hz and then sends them all to the gateway station (a PDA) for processing. The gateway station uses these data as input to compute the heart rate, SpO2 and blood pressure. These outputs will be sent to the clinical department for further analysis. In the event of an emergency, an alarm will be raised.

As the ECG and SpO2 sensors and their ADC were not available in the repository, we have to model them first. The module provides an interface named `PpgSensor`. After constructing the `ppgSensorC` module, we have all the necessary modules. We then use a UML profile for TinyOS to construct the ECGnSpO2 application.

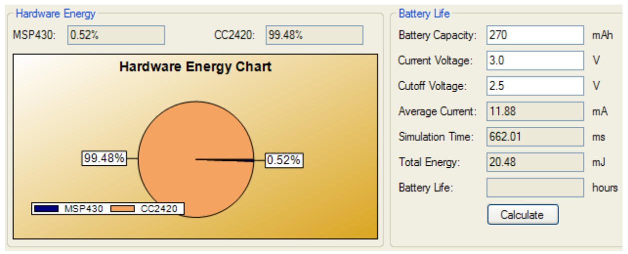
V. Experiment Results

For both our case studies, we drew the UML diagrams using IBM Rhapsody. UML2nesC, a Java program, performs the generation of the BSN application in of nesC and UML2SystemC generates the SystemC based BSN simulator. The generated nesC code was successfully deployed on the BSN motes from Imperial College [15] as well as the generated BSN simulator. The simulator is cycle-accurate and therefore, models the real system closely. Table II shows the application size and execution time of the nesC code generator. In our designs, existing UML models were reused. For example, when we designed the ECGnSpO2 application, part of the structure model was taken from the wheeze detection application.

Table II
Code size and execution time of our automatic code generator.

Application	**Lines of generated nesC code**	**Time taken for code generation**
WheezeDetection	361	0.59 second
ECGnSPO2	1046	1.1 second

With the aid of simulator, we are able to estimate the timing and energy consumption without any real hardware. The wheeze detection application has a hard real time deadline of 32 ms to complete a detection cycle. The existing hardware platforms could not meet this timing requirement, and a new sensor node is currently under development with twice the processing power. To account for this, we adjusted the clock rate in the simulator, and the software development and tuning continued even as the hardware was being built. Finally, we reduce the execution time to process each data block from 46ms to 23ms in `WheezeDetection` module. For the ECGnSpO2 application, energy consumption was the main concern. By executing our ECGnSpO2 implementation, we were able to obtain the detailed energy profile. We found that wireless transmission takes up more than 90% of the total energy consumption. The original implementation consumed 39.58mW. With the proper adjustments, we could reduce the power consumption to 30.94mW (mJ/ms) shown in Figure 5.



MSP430:	0.52%	
CC2420:	99.48%	
Battery Capacity:	270	mAh
Current Voltage:	3.0	V
Cutoff Voltage:	2.5	V
Average Current:	11.88	mA
Simulation Time:	662.01	ms
Total Energy:	20.48	mJ
Battery Life:		hours

Figure 5. Experiment result of lower power consumption 30.94mW (20.48mJ/662.01ms) for the ECGnSpO2 application by proper ajustment.

VI. Conclusions

In this paper, we presented a UML 2-based framework that aims to assist designers in managing the complexity and reusability of TinyOS-based BSN applications. Together with our UML-modeled, automatically generated simulation environment, we have a design flow that not only allows for the co-design but also the co-optimization of BSN applications on possibly as yet non-existent hardware by estimating time and energy characteristics of the platform. Case studies of actual BSN applications provide evidence to this claim. Furthermore, they show the efficiency of the code produced by our generator.

References

[1] G. Yang, *Body Sensor Networks*. Springer-Verlag New York, Inc, 2006.
[2] D. Gay *et al.*, "The nesc language: A holistic approach to networked embedded systems," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, 2003.
[3] P. Levis *et al.*, "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Springer Berlin Heidelberg, 2005, pp. 115–148.
[4] UML profile specification. [Online]. Available: http://www.omg.org/technology/documents/profile_catalog.htm
[5] L. Lavagno, G. Martin, and B. Selic, *UML for Real: Design Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003.
[6] N. Kathy *et al.*, "Model-driven SoC design: The UML-SystemC bridge," in *UML for SOC Design*, G. Martin and W. Mller, Eds. Springer US, 2005, pp. 175–197.
[7] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "A SoC Design Methodology Involving a UML 2.0 Profile for SystemC," in *Proc. IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, 2005, pp. 704–709.
[8] A. Varga, "The OMNeT++ discrete event simulation system," in *in Proceedings of the European Simulation Multi conference (ESM2001)*, Prague, Czech Republic, June 2001.
[9] H. Tyan, "Design, realization and evaluation of a component-based compositional software architecture for network simulation," Ph.D. dissertation, Ohio State University, 2002.
[10] L. Girod *et al.*, "Emstar: A software environment for developing and deploying wireless sensor networks," in *in USENIX 2004 Annual Technical Conference*, 2004, p. 283296.
[11] Aurora. (2007) The AVR simulastion and analysis framework. [Online]. Available: http://compilers.cs.ucla.edu/avrora/
[12] GRATIS. [Online]. Available: http://w3.isis.vanderbilt.edu/Projects/nest/gratis/GratisIITechOver.html
[13] I. Cutcutache *et al.*, "BSN Simulator: Optimizing application using system level simulation," in *Proceedings of the 6th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2009)*, Berkeley, CA, U.S.A., June 2009, pp. 9–14.
[14] F. E. Tay, D. Guoa, L. Xua, M. Nyana, and K. Yap, "MEMSWear-biomonitoring system for remote vital signs monitoring," *Journal of the Franklin Institute*, vol. 346, no. 6, pp. 531–542, August 2009.
[15] BSN node specification. [Online]. Available: http://vip.doc.ic.ac.uk/bsn/index.php?article=167