UML-BASED CO-DESIGN FRAMEWORK FOR BODY SENSOR NETWORK APPLICATIONS

SUN ZHENXIN

NATIONAL UNIVERSITY OF SINGAPORE

2011

UML-based Co-design Framework for Body Sensor Network Applications

Sun Zhenxin

(B. Computing(Hons), National University of Singapore, Singapore)

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE NATIONAL UNIVERSITY OF SINGAPORE

Acknowledgement

First of all, I would like to express my deepest gratitude to my supervisor, Prof. Wong Weng-Fai, for his patience, support and help all these years. I have received immense support both in academics and life from him. Without his professional guidance and inspiration, this thesis would not even be possible.

I am deeply grateful to Prof P.S Thiagarajan, for his detailed and constructive comments on some of my works.

My sincere thanks are due to Kathy, Nguyen Dang for who have been collaborator of some initial works, and co-author of some of my papers. It was great pleasure to work with her.

I would like to thank my labmates in Embedded System Lab, who have been very kind and supportive in my research life: Joon, Edward Sim, Pan Yu, Andrei Hagiescu, Wang Chundong, Ge Zhiguo, Ankit Goel. My graduate life at NUS would not have been fun and interesting without them.

Special thanks go to National University of Singapore for funding me my research scholarship. My thanks also go to administration staff in School of Computing for their supports during my study.

My deepest appreciation goes to my family. My parents gave me much love, and their encouragements are my great source of power. I owe my loving thanks to my wife Wang Yulian, and my daughter, Sun Wanqing, and my son Sun Qichen, whose love

enables me to overcome the frustrations which occurred in the process of writing this these. This thesis dedicated to them.

Finally, I would like to thank all people who have helped and inspired me during my doctoral study.

TABLE OF CONTENTS

ABSTRACT

A *body sensor network* (BSN) refers to a set of communicating, wearable computing devices. They are gaining popularity especially in bio-monitoring applications. In body sensor networks, the hardware and software often need to be co-designed specifically for an application. BSN applications are particularly sensitive to the tradeoff between performance and energy, both of which are also often severely constrained. However, often the hardware is still under development when the application running on it is being implemented. This makes any estimation of this tradeoff in the design of the hardware and the software inaccurate. In this thesis, we propose a unified design framework to manage the complex development of BSN application with the aims of enhancing modularity and reusability.

The proposed framework consists of a set of Unified Modeling Language (UML) 2.0 profiles for both software and hardware designs. For software portion, we have chosen to use nesC-TinyOS, the most popular programming language (nesC), and runtime system (TinyOS) platform for BSN applications. For hardware, we have chosen to use SystemC, the de facto standard specification language for hardware design. The proposed UML profiles abstract the low-level details of the application, and provide a higher level of description for application developers to graphically design, document and maintain their BSNs that consist of both hardware and software components. Using profiles for hardware platforms, we are able to customize a UML-based hardware simulator for BSNs. Our interface synthesis technique allows us to reuse existing design components (IPs) with a “plug-and-play” approach. This highly-

reconfigurable simulator acts as a fast and accurate performance evaluation tool to aid both software and hardware design.

With the aid of a UML profile for TinyOS and a pre-defined component repository, minimum knowledge about TinyOS is needed to construct a body sensor network application. The hardware simulation environment allows users to customize the hardware platform before a commitment is made to the real hardware. In our framework, we have also modeled our simulator in UML. Customized simulator can be automatically generated after refining system model or re-configuring the hardware parameters. Key design issues, such as timing and energy consumption can be tested on this automatically generated simulator. The framework ensures a separation of software and hardware development while maintaining the close connection between them.

The thesis will describe the design and implementation of the proposed framework, and how the framework is used in the development of nesC-TinyOS based body sensor network applications. Actual cases studies are used to show how the proposed framework can be used to adapt quickly to changes in the hardware while automatically morphing the software implementation quickly and efficiently to fit the changes.

LIST OF FIGURES

List of Tables

List of Publications

1. K.D. Nguyen, Z. Sun, P.S. Thiagarajan, and W.F. Wong, "Model-driven SoC Design Via Executable UML to SystemC", Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS). pp. 459-468. Dec 2004
   (Chapter 2)
2. Y. Zhu, Z. Sun, A. Maxiaguine, and W.F. Wong, "Using UML 2.0 for System Level Design of Real Time SoC Platforms for Stream Processing". Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 154-159. Hong Kong. Aug 2005.
   (Chapter 2)
3. Z. Sun, Y. Zhu, W.F. Wong, and S.K. Pilakkat, "Design of Clocked Circuits using UML". Proceedings of the Asia and South Pacific Design Automation Conference 2005 (ASP-DAC)." pp. 901-904. Jan 2005
   (Chapter 2)
4. K.D. Nguyen, Z. Sun, P.S. Thiagarajan, and W.F. Wong, "Model-Driven SoC Design: The UML-SystemC Bridge" in "UML for SOC Design" edited by Grant Martin and Wolfgang Müller. pp. 175-197. ISBN 0-387-25744-6. Springer. July 2005
   (Chapter 2)
5. Xianhui He, Yongxin Zhu, Zhenxin Sun, Yuzhuo Fu: UML Based Evaluation of Reconfigurable Shape Adaptive DCT for Embedded Stream Processing. EUC Workshops 2006: 898-907
6. Cutcutache, T.T.N. Dang, W.K. Leong, S. Liu, K.D. Nguyen, L.T.X. Phan, E.J. Sim, Z. Sun, T.B. Tok, L. Xu, F.E.H. Tay, and W.F. Wong, "BSN Simulator: Optimizing Application Using System Level Simulation", Proceedings of the 6th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2009), pp. 9-14. Berkeley, CA, U.S.A., Jun 2009.
   (Chapter 5)
7. Z. Sun, and W.F. Wong, "A UML-Based Approach for Heterogeneous IP Integration", Proceedings of the 14th Asia and South Pacific Design Automation Conference (ASP-DAC)." pp. 155-160. Yokohama, Japan. Jan 2009
   (Chapter 3)
8. Z. Sun, C-T. Ye, and W.F. Wong, "A UML 2-based HW/SW Co-Design Framework for Body Sensor Network Applications". Proceedings of Design, Automation, and Test in Europe (DATE 11). pp. 1505-1508. Grenoble, France. Mar 2011
   (Chapter 5 and 6)

Chapter 1 Introduction

1.1 **Body Sensor Network**

With the recent advances in wireless sensor network and embedded computing technologies, wearable sensing devices have become feasible in meeting the demands for healthcare and bio-monitoring. There are now numerous examples of such body sensor network (BSN) applications [25][34]. Generally speaking, a BSN system collects, processes, and stores physiological (such as electrocardiogram (ECG), and blood pressure), activity (such as walking, running, and sleeping), and environmental (such as ambient temperature, humidity, and presence of allergens) data from a host's body and its immediate surroundings. It may even be able to perform actuation (such as in the form of drug delivery) based on the data collected[10]. BSNs can be very useful in assisting medical professionals to make informed decisions about the course of the patient's treatment by providing them with continuous information about the patient's condition.

Due to its reliability and the ability to detecting the onset of acute diseases, or monitoring chronic illnesses, BSNs have been used in a wide range of applications. The common and important applications of BSN include the monitoring of patients who have left hospital care, or the detection of the onset of different conditions such as asthma, or heart attacks. The following are two examples of sound BSN applications:

- A BSN worn by a patient that automatically alerts the hospital at the critical moment just before the onset of a heart attack, through measuring changes in the patient's vital signs.

- A BSN on a diabetic patient that automatically injects insulin through a pump, as soon as the level of insulin falls below a certain critical level.

Other applications of this technology include sports, military, and security where there is a need for the seamless exchange of information between individuals, or between individual and machines.

1.2 Conventional Design Methods

Given the wide variety of BSN applications, each with its own customized hardware and software, the design of these applications is attracting more concerns. The basic structure of a BSN is shown in Figure 1. As shown in the figure, wireless wearable sensors and a *local processing unit* (LPU) are attached to human body to measure the status of the patient. The latter is normally a PDA or a smart phone. The sensors collect information, and after doing some basic processing, they will send the results to the LPU via wireless communication channels. The LPU processes the data collected from sensors, and makes decisions such as sending alert, or making a call to the doctor in charge.

Figure 2 outlined the conventional design flow for BSN applications. It usually starts with some informal design specifications. These specifications include functional and performance requirements. Designers will then partition the BSN system into software and hardware portions, and assign them to different developers. The hardware and software designers will then work independently based on these specifications. Hardware components are refined from the specifications to the implementation.

Software components are subsequently implemented and integrated with the hardware components to complete the system. It is only after this integration, the functionality and performance metrics can be evaluated. If bugs are found at this stage, identifying and fixing the bugs is tedious and time-consuming. A large amount of effort generally needs to be spent at the post-developing stage.

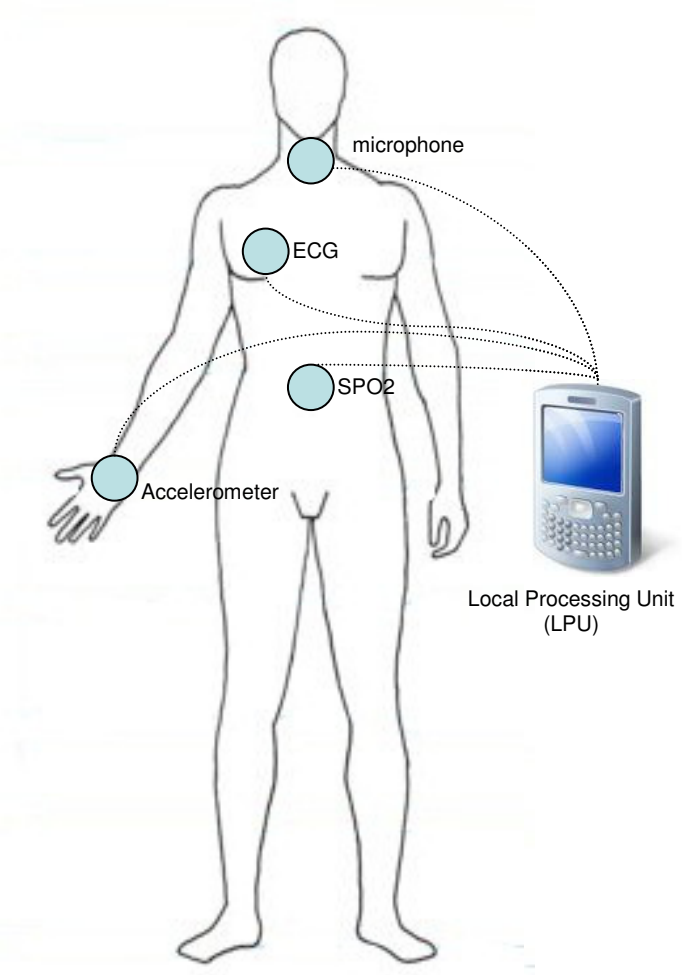


Figure 1: A typical Body Sensor Network design

Integration puts together the separate software and hardware components to form the complete system. It is also a time consuming and error-prone process. The separate development of hardware and software restricts the ability to study the tradeoffs between hardware and software. A "hardware first" approach is often pursued with the following consequences:

- Hardware is specified without a full understanding the computational requirements of the software.
- Software development does not influence hardware development, and changes made to hardware may not be reflected promptly in software.

Following this approach, any problem encountered as a result of late integration can cause costly modifications and schedule slippage.

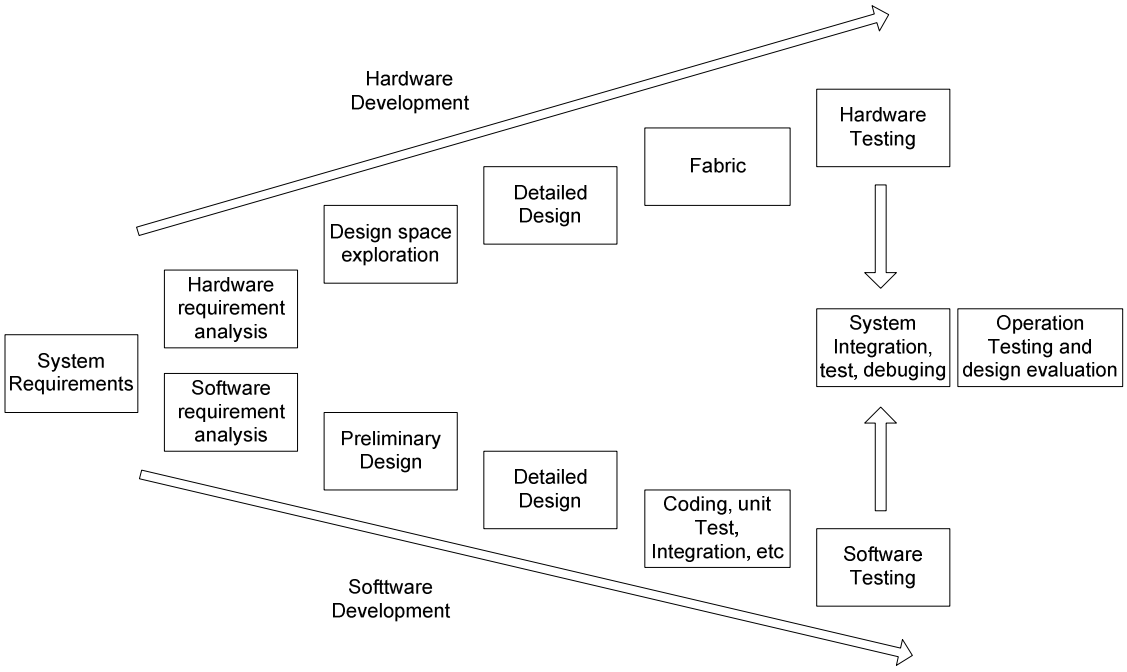


Figure 2: A traditional design flow of BSN applications

1.3 Challenges of Body Sensor Network Application Design

Developing BSN systems is not easy. The nature of BSN system brings unique challenges to BSN designers. BSN applications have to operate in a continuous manner on the host body. Much of the theoretical foundation for general wireless sensors also applies to BSNs, with particular emphasis on issues such as power optimization, battery life performance, and radio design. There are also other issues, such as usability, durability, robustness, how well the sensor 'fits' in with the application, and the reliability and security of the data, that must be considered for a successful deployment of any BSN system. Sensor networks often suffer from the so-called 'reliability dilemma' -- the more reliable and secure is the data transmission, the higher is the overhead, and consequently more power is required. This leads to a reduction in battery life which is generally a bad thing for BSNs.

Another of the frequent issues that a BSN designer encounters is that hardware components have to be customized to meet any new design requirements. Continuous monitoring requires a lot of energy, and to give accurate and immediate detection result, a BSN node has to meet processing speed and transmission rate requirements that are often very energy consuming. On the other hand, BSN devices must be small enough to be wearable or even implanted, hence limiting its computational power and energy. This challenges the BSN designer to find an optimal design that can achieve the right balance of battery life and accuracy of measurement. The challenge for the designer would be how he can use these components in such a way that the stringent requirements of the application are met. There are also standard BSN devices available.

The designer also has the flexibility to customize the BSN hardware components, for example, increasing the bit width, or adjusting the clock's speed when trying to match the computational accuracies and energy requirements. Given the large design space, searching for the optimal solution may be very time-consuming.

To achieve the optimal solution, designers need to explore the different design alternatives and tune their application constantly. The often non-availability of real hardware is a particularly serious challenge. Because the target hardware and the software are being developed concurrently, BSN application designers have to rely on existing BSN platform to verify and test their implementation. However, differences in the hardware make the timing and energy analysis inaccurate, and the implementation will require another round of customization before the final integration with new hardware platform. A flexible and efficient tool for pre-integration testing would be useful.

In BSN application, hardware and software components have greater impact on each other than in other platforms. However, without actual integration, the exact nature of this dependence becomes very difficult to characterize. Continuous assessment of the overall performance metrics will help designer find the optimal solution more efficiently.

Moreover, the late integration also postpones the evaluation of design metrics and bug identification, such as bugs in communication channel which may only be found after integration. Identifying and fixing a bug after integration can be very difficult and expensive for both hardware and software designers. Figure 3 shows the

recommended design flow for BSN application designs is supported by our proposed framework.

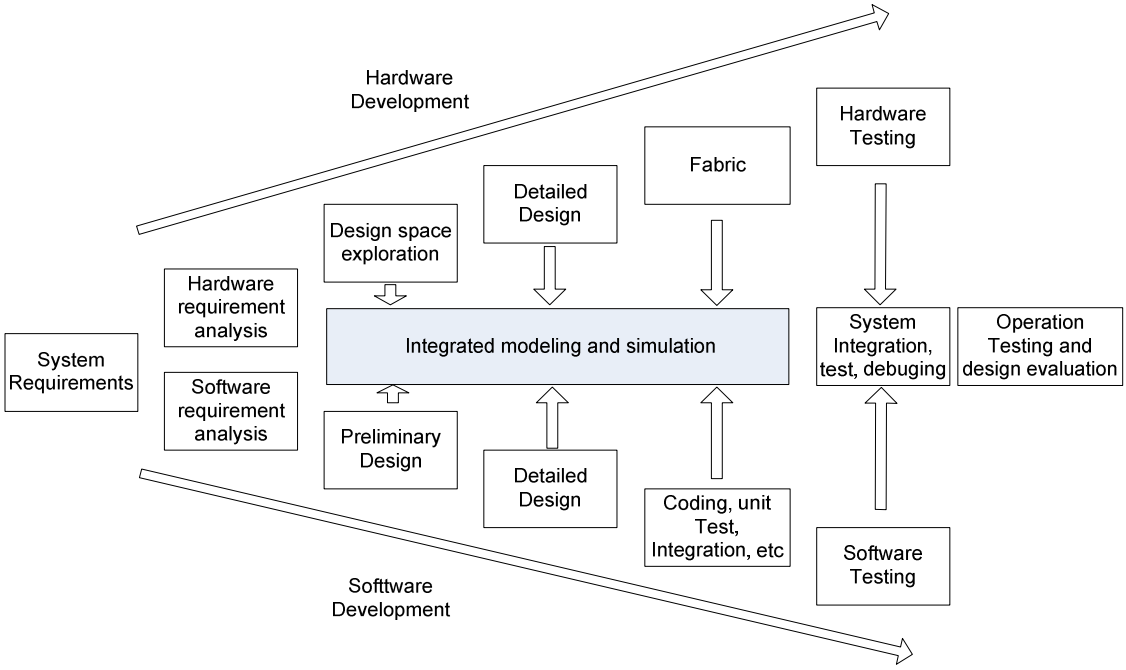


Figure 3: Our recommended design flow for BSN application

1.4 **Embedded System Design and UML**

A BSN application is a kind of embedded system. Therefore the design techniques for embedded systems also apply to BSN. In this section, we will elaborate the design and co-design techniques using UML that is inherited from general embedded systems.

An *embedded system* is a *computer system* designed to perform specific functions often with real-time computing constraints. It is embedded as part of a more complete device typically including hardware and software parts. The uses of embedded systems are virtually limitless, because every day new products are introduced to the market that

utilizes embedded computers in novel ways. Products with embedded systems include computer parts, mobile phones, machine control units, automobile, etc, and they are closely related to our everyday life. All of these makes embedded system evolve very rapidly. On the other hand, embedded system designs are performed in a rather ad hoc way. Different types of algorithms (e.g. signal processing, communications, and controls) are implemented using a variety of technologies (e.g., digital signal processors, microcontrollers, field-programmable gate arrays, application-specific integrated circuits, and real-time operating systems). The complexity and scale of embedded systems bring many challenges to embedded system designers.

With advances in semiconductor technology, it is now possible to integrate most if not all the hardware modules required by an embedded systems into a single chip, giving rise to the so-called “system-on-a-chip” (SOC) platform. With complexities of SOCs rising rapidly following Moore’s Law, the design community has been searching for new methodologies that can handle the complexities with increased productivity and decreased times-to-market. The obvious solution that comes to mind is increasing the levels of abstraction. In other words, using a modular approach to compose the overall system with increasing larger basic building blocks (or “intellectual property” (IP) blocks). However, it is not clear what these basic blocks should be beyond the available processors and memories. Furthermore, with multitude of processors and variety of IP blocks on the chip, the difference between software and hardware design has become indistinguishable. However, the existence of often incompatible tools for

utilizing and deploying the hardware and software components has prevented the creation of a comprehensive and integrated design flow.

Hardware and software essentially share the same development pattern. With dramatic increases in the size and complexity of both hardware and software, more user friendly and reusable design and development methodologies were introduced to cope with these issues.

Introducing the Unified Modeling Language (UML) into the design of system-on-chip has become an accepted solution to the ever increasing complexity. UML is a standardized general specification language which was originally created for software-based system. For decades, software designers have been applying it to every stage of software design and implementation. With the power of the UML, designers are able to model their application from different points of view, share their ideas with commonly understood notations.

A key strength of UML is its ability to be extended with domain-specific customizations, as so-called *profiles*. A profile in the Unified Modeling Language (UML) provides a generic extension mechanism for customizing UML models for particular domains and platforms. Extension mechanisms allow refining standard semantics in strictly additive manner, so that they can't contradict standard semantics. Profiles are defined using stereotypes, tag definitions, and constraints that are applied to specific model elements, such as Classes, Attributes, Operations, and Activities. A Profile is a collection of such extensions that collectively customize UML for a

particular domain. In this thesis, we will propose a few UML profiles to model BSN applications in different domains.

By decoupling the specification from implementation and using formal mathematical models of computation for specification, we gain the ability to perform fast simulation and efficient synthesis of complex heterogeneous systems. We model complex systems as a hierarchical composition of the simpler models of computation. Some of these simpler models of computation, such as types of finite state machines, dataflow models, and synchronous/reactive models, have finite state. Because they have finite state, all analyses of the system can be performed at compile-time. For example, memory usage and execution time can be determined without having to run the system. These models can be overlaid on an implementation technology (such as C or VHDL).

1.5 **Levels of Abstractions**

Design of anything, from a web application to an embedded software system to a hardware device, is done at some level of abstraction. Simply, a level of abstraction is the term that the designer uses to describe the thing being built. Level of abstraction usually refers to the level of complexity by which a system is viewed or programed. The higher the level, the less detail. The lower the level, the more detail. A level of abstraction is determined by the objects that can be manipulated and the operations that can be performed on them. In programming terms, the objects are data types and the operations are the operators that can be used in expressions and control constructs.

The notion of abstraction levels was introduced with software systems, since there were several obvious levels: machine code, assembly language, subroutine libraries, and eventually interpreted application languages like SQL. In hardware design, we are familiar with several low levels of abstraction: polygon, transistor, gate, and register-transfer (RTL). On top of these layers, hardware designers have introduced a few more levels to cope with the ever increasing complexity. Figure 4 gives a list of typical levels of abstraction being used in software/hardware design. The levels from top to bottom are: Application level, assembling level, machine code level, hardware level, transaction level(TLM), behavioral level, Register transfer level(RTL) and netlist level, and gate level.

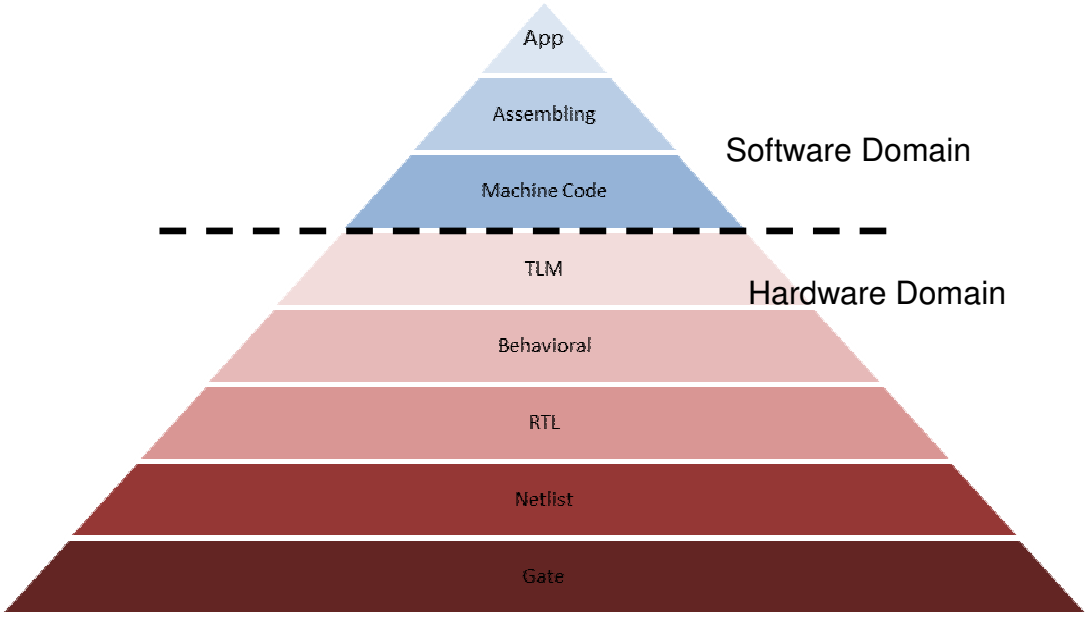


Figure 4: Typical levels of abstraction in software/hardware design

Gate level

At gate level, hardware is described as connected logic gates or transistors. The combinations of these gates will react at the edge of clock and perform the logic functionalities.

Netlist

In Netlist level, hardware systems are described as directed graph with simple boolean gates as nodes.

Register transfer level

RTL level code is characterized by arithmetic expressions, including conditionals and control constructs, which execute in a fixed schedule of cycles. The data objects, registers and wires, typically correspond to hardware objects. Low level details such as clock signals are also modeled. RTL simulation is normally defined as cycle accurate.

Behavioral

Behavioral code looks like normal sequential code found in a general purpose programming language. It does, however, have structural information in the form of modules and ports. Simulation at behavior level can be defined as cycle accurate, and the simulation is faster than RTL level.

Transaction level

A transaction object is an abstraction of an interface between modules (or more generally, concurrent processes). An example would be a FIFO buffer. The transaction object would take the place of several ports in the module port list. The operations done

on the object would be get and put, and perhaps query status. Another example would be an AMBA bus interface. Simulation at transaction level is usually faster than the previous levels and not cycle accurate.

These levels of abstractions are used in different stages of the hardware design process. Gate level and netlist level are very close to the real hardware, and they are used by circuit designers for physical designs. Hardware designers spend great effort on system level designs, which is at RTL and higher levels. Simulation speed varies in at great scare. For example, the RTL level simulation can be 100 time slower than the simulation at TLM level models. However, simulation at RTL level is defined as cycle-accurate, which is much more accurate comparing to the transaction level. The lower level simulation will take even more time. Our design framework will lift the abstraction level up to UML level, and designer will be able to focus on the relatively high abstraction levels, starting from UML level and evolves to the register transfer level. Our research is focusing on system level designs, although the produced implementations might require further optimization in physical design procedure. The main objective of the UML framework is to help the designers on refinement of codesign system with fast prototyping and integrated simulation models.

Our framework will focus on system level design procedures where simulation models are used to describe the design. In our framework, model simulation is the main method for the system analysis and validation. Transitions are made automatically from higher level to lower level with minimal user interaction. By working at a high level of abstraction, our framework can be used earlier in the design process than existing tools.

We automatically generate the working code which can be used directly for simulation, and through simulation, we can obtain accurate evaluation of the system, and therefore the designers can optimize their design based on the simulation result.

1.6 Our Contributions

This thesis summarizes our research on exploring usage of UML on BSN design. The main contribution of this thesis is the unification of software and hardware design using a single design platform: UML, and subsequently applying it to the design of BSN applications. The detail contributions are as follows.

- We explored how UML can be used to design the digital components of the embedded system. We used UML to capture functional and behavioral specification of what are low level designs. A mapping was established between UML and a hardware description language, namely SystemC. Different levels of executable implementation can be automatically generated to perform simulation and verification. The semantics of chosen UML notations are then been extended to capture more design constraints such as clocking.

- We extended the UML profile for digital hardware to also handle analog and mixed signal systems. Many sensors are analog in nature, and their output need to be translated by mixed signal components for digital processing and communication. In order to express analog and mixed signal designs, we have chosen to use SystemC-AMS, a SystemC-complaint hardware description language for analog systems, as the implementation language. In a similar way,

UML is used to capture the specifications of mixed signal systems. Mappings are established between the design and its implementations. Using this novel approach, the analog part of the application can be seamlessly integrated with the digital components to capture hardware designs with a high degree of fidelity.

- We introduced a UML-based interface synthesis technique that solves the problem of integration heterogeneous "intellectual property" (IP) blocks by reconfiguring their bus interfaces. An IP block refers to a pre-defined hardware or software functional component that has a well-defined interface. IP blocks may perform similar functionalities but their interfaces may differ, and are normally incompatible with the current design. Reusing existing IP blocks for a new design makes a lot of economic sense as they already have been purchased, or designed and tested. However, the incompatible interfaces make reuse technically challenging. To facilitate reuse, we chose the OCP (Open Core Protocol) as the standard bus interface. We use UML to model the IP communication interfaces, and automatically generate wrappers, so that the packaged IP cores can be easily assembled together using the OCP bus standard for simulation. In order to do this, we extracted the essential aspects of the bus communication, and modeled the interfaces using UML notations. UML notations are then used to produce wrapper code. The generated wrappers can be directly used to wrap up IP blocks for test and simulation.

- We developed a customizable simulator for BSN hardware platforms using the proposed hardware profiles and interface synthesis technique based on UML. Exiting IP blocks can be reconfigured and added in to the simulation platform, and our interface synthesis technique allows us to reuse existing design component (IPs) with a "plug-and-play" approach. The hardware simulation environment allows users to customize the hardware platform before a commitment is made to real hardware. In our framework, we have also modeled our simulator in UML. Implementations can be automatically generated by simply configuring some hardware parameters. Key performance issues, such as timing and energy consumption can be tested by simulating the generated implementation on this automatically generated simulator. This highly-reconfigurable simulator provides a fast and accurate performance evaluation tool to aid both software and hardware design.
- We proposed a UML profile for the software portion of BSN applications, and we have chosen TinyOS as our target platform. TinyOS is the most popular software platform of BSN application. The program running on TinyOS is written in nesC, a dialect of C. Using this UML profile, a designer can focus on the high level specifications rather than worry about the low level nesC implementation during the design stage. The models of the TinyOS components are kept in a repository. We find that the BSN applications often share similar structures, and these models are highly reusable. With the aid of a UML profile for TinyOS and a pre-defined component repository, minimum knowledge of TinyOS is needed to construct a body sensor network system.

- We proposed a novel co-design framework for BSN applications that is based on UML. The proposed UML profiles abstract the low-level details of the application and provide a higher level of description for application developers to graphically design, document and maintain their systems that consist of both hardware and software components. The framework ensures a separation of software and hardware development while maintaining the close connection between them.

The thesis will describe the design and implementation of the proposed framework, and how the framework is used in the development of nesC-TinyOS based body sensor network applications. Realistic cases studies are used to show how the proposed framework can be used to adapt quickly to changes in the hardware while automatically morphing the software implementation quickly and efficiently to fit the changes.

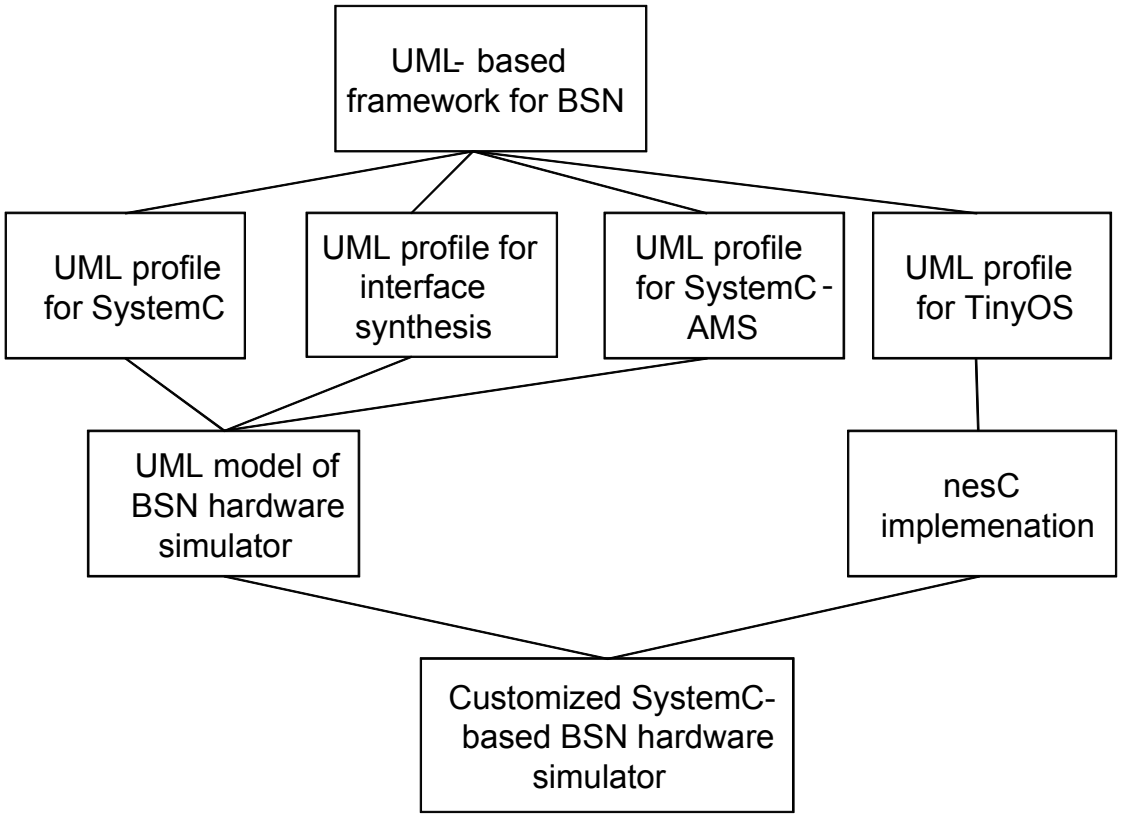


Figure 5: Structural diagram of our BSN co-design framework

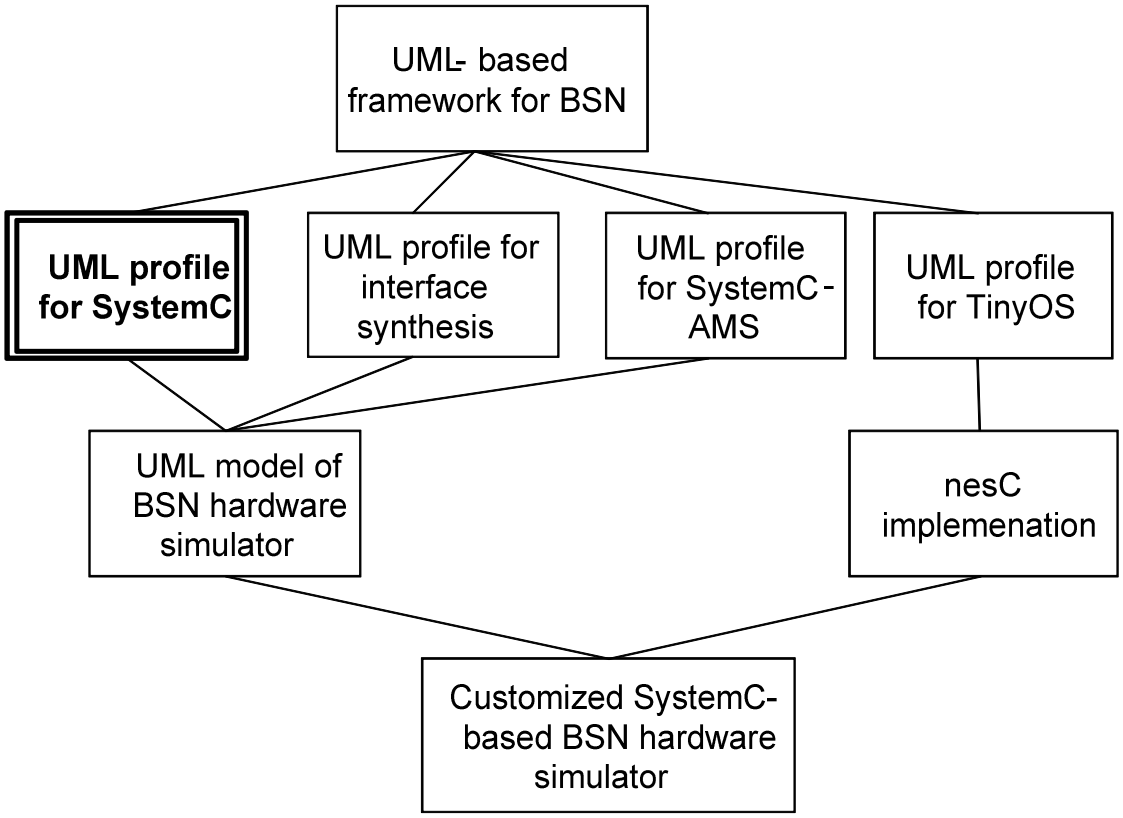
1.7 **Thesis Structure**

Figure 5 shows the structural contents of the thesis. In the structures, several UML profiles are the directly contributions. Based on these profiles, we apply them to aid BSN software and hardware design and implementation.

Reminder of thesis is organized as follows. In next chapter, we will present UML profile for SystemC which are used to model the digital portion of an application. Some related works, such as UML, SystemC, will also be introduced in chapter 2. UML-based interface synthesis technique is presented in chapter 3, which will be used to specify and generate glue logic between pre-defined hardware components. In chapter

4, we outline the UML profiles for SystemC AMS which can be used to model the analog portion of system. In Chapter 5, we will apply the profiles to a BSN hardware simulator, the details of UML profiled SystemC simulator for BSN hardware platform will be presented. In chapter 6, we first present the design framework for TinyOS, and then we will show how we can unify the software and hardware portion of BSN application by integrating the generated software code with the SystemC based simulator. Chapter 7 concludes the thesis and presents possible directions for future works.

Chapter 2 Background and Related Works on our UML-based Framework



2.1 UML

With the increasing complexities of embedded systems, designers have been searching for new methodologies that can manage the complexity as well as yielding high productivity. The Unified Modeling Language (UML) is a proven modeling and specification language that has been used widely in development of complex software applications [80]. Embedded designer found that embedded system design can benefit from UML in similar way.

UML is an object-oriented modeling language standardized by the Object Management Group (OMG) mainly for software systems development[80]. It is a visual modeling tool for specifying, visualizing, constructing and documenting software systems and business processes. UML consists of a set of basic building blocks, rules that dictate the use and composition of these building blocks, and common mechanisms that enhance the quality of the UML models. Its rich notation set has made UML a popular modeling language in multiple application domains for system documentation and specification, for capturing user requirements and defining initial software architecture. It is considered the best understanding of system by designers and programmers.

While UML is well-suited for modeling software systems in general, it lacks support for some aspects important to embedded real-time systems, e.g. modeling of timing constraints, signals, and independent components[97]. Therefore, different proposals to extend the UML for modeling real-time systems have been made. The

Object Management Group (OMG) proposes to extend the UML by building UML "profiles" that contain the needed extensions[81]. However, this extension mechanism is currently not part of the standard and it is still discussed how to realize it. In parallel the leading CASE tool vendors implement proprietary extensions to the UML. Rational and Telelogic adapt UML for modeling embedded real-time systems by combining it with the modeling languages from the real-time (ROOM) and telecommunication domains (SDL), while I-Logix stays with Standard-UML, but provides a very powerful implementation of statecharts. [68]

The current version of UML called UML2, which is a large collection of diagrams and notations. It has 13 diagrams types. Here, we briefly introduce the diagrams we use in our framework. UML class diagram and state chart are chosen to capture the structure, behaviors and the deployment of the hardware components.

Class diagrams show the building blocks of the system, their inter-relationships, and the operations and attributes of the classes. We found that class diagram can represent the structural information of embedded system in a quite natural way. The individual components of the hardware components can be drawn as classes, and wiring among them can be modeled as relations or interfaces bindings via UML port. Stereotypes can be used to distinguish the difference types of hardware components. The composite classes enables designer to view the under design system from system level view down to detailed implementation of a building class. Each of the class in class diagram can be associated with a state chart diagram to specific its behaviors. State charts depict the dynamic behavior of an entity based on its response to events,

showing how the entity reacts to various events depending on the current state that. This event-triggered model is well-suited abstraction to the signal-triggered hardware components, where signals can be considered as events, and output signals are modeled as reactions. From practice, we found that these two types of diagrams are well-suited to our specification requirements.

2.2 SystemC

SystemC[75] is a system level modeling language based on C++. It provides library supporting system level design. It has become de facto standard for embedded hardware design language. SystemC has desirable properties for system level design. Besides, SystemC use most of C++ grammar, and this allows user to learn it in a very short time. Even those who have no experience with programming in SystemC can read and understand the code.

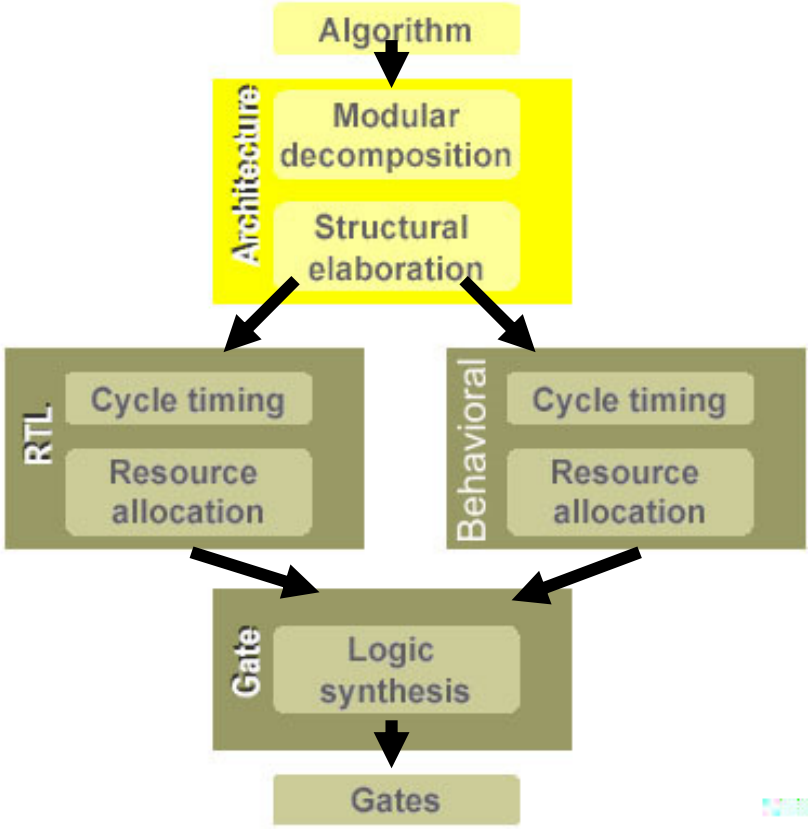


Figure 6: Different levels of SystemC model

Using the SystemC library, designer can model a system at various levels of abstraction as shown in Figure 6. At the highest level, only the functionality of system may be modeled. For hardware implementation, model can be written either in a functional (behavioral) style or RTL (register-transfer level) style. [40] The software part of a system can be naturally described in C++. Interfaces between software and hardware and between hardware blocks can be described either at the transaction-accurate level or at the cycle-accurate level. Moreover, different parts of the system can be modeled at different levels of abstraction and these models can co-exist during system simulation. C++ and SystemC classes can be used not only for the development

of the system, but also for the test-bench. SystemC consists of a set of header files describing the classes and a link library that contains the simulation kernel. Any ANSI C++ compliant compiler can compile SystemC, together with the program. During linking, the simulation kernel of the SystemC library is used. The resulting executable serves as a simulator for the system designed.

SystemC provides an ideal platform for developing embedded systems. Software and hardware parts can both be specified using the same language and verified using a common test-bench. The hardware parts may be refined up to RT level and implemented by using synthesis tools. The hierarchical modeling features of SystemC are supported by the hierarchical specification model. This facilitates not just a structured design, it also enable IP reuse. The FSMs can also be organized in a hierarchical manner, implementing a Hierarchical control flow.

SystemC-AMS[76] is an extension of SystemC[75] to describe mixed-signal design, and it has been popularly studied to model mixed signal systems such as an inertial navigation system [18], the I2C protocol communication [50], and wireless sensor network node [54]. We will focus on SystemC-AMS profile in chapter 3.

2.3 UML-based SystemC Design Methodology

This section briefly discusses other projects that have investigated integrating formal and informal approaches to systems development, where multiple modules are used to describe a system.

2.3.1 Our previous works on UML to SystemC

Some of our previous works was focusing on the UML-based design technique on hardware[37][38][93][97]. In these works, we have proposed a UML profile for SystemC to capture the design specification of a system, including its architecture and behaviors. Two types of UML diagrams, namely class diagram and statechart diagrams are chosen as the modeling tool. With proposed profile, designers can leverage the design abstractions to UML level, and with the designed models can be translated into SystemC executable programs to do simulation so that designer can verify whether the design satisfies the requirements based on the simulation results. If the results are not satisfactory, designers can go back to UML model and modify it, re-generate SystemC code and check its behaviors again.

The UML models can be mapped into different levels of abstraction for different purposes. We have supported transaction level modeling(TLM), behavioral and register transfer level(RTL). TLM level offers faster simulation rate, and the BSN simulator presented in chapter 5 will employ this level of simulation. Designer can also target the code generation to RTL level, which describe the operation of a digital circuit with hardware registers and signals, the RTL level components can be considered the program running on the target platform.

2.3.2 YAML

Most of the effort we have seen in the UML-SystemC translation was to generate skeleton SystemC code from, in particular, class diagrams and object diagrams. An

example of this kind of projects is YAML[84]. YAML uses UML notations to model hardware and allows user to input information about objects and relationships into a UML class diagram (for behavioral hierarchy) and object diagram(for structural modeling). YAML generates the C++ code for the design, using information input by the user to the UML class and object diagrams.

YAML provides a user friendly graphical interface to model systems under the guidelines of UML, using the SystemC and ICSP C++ class libraries. Users can specify the details of SystemC and ICSP classes into the UML front end. The code generated by YAML conforms to the syntax of ICSP and SystemC classes and can be directly compiled and simulated.

The major advantage of using YAML is the ability to avoid the complex syntactic details involved in using the C++ libraries. User can generate the SystemC + ICSP code from YAML, after specifying the various details in the class and object diagram. YAML has been used to model various designs including a DLX compatible processor pipleline. The DLX pipeline code consists of around 2000 lines of C++ code. Most of which was generated automatically by YAML. It raises good results in generating and simulating the models.

YAML gives some ideas of modeling the system functional and structural information. However, it lacks of the behavioral requirement support, and hence cannot capture the requirement of control information.

2.3.3 Auto-generation of SystemC model from Extended Task Graphs

To model the behavior of hardware, Klaus proposed to use extended task graph[70]. Task graph gives an accurate definition of time and different model of computation are emphasized. Task graphs are a widely-spread means for the specification of embedded systems behavior. Task graphs have a well-defined execution semantic and a temporal order and other abstract modeling characteristics. A task graph represents operations and data dependencies between them. Its main features are both the modeling of control flow and a hierarchical structuring of functionality.

The methodology was successfully applied to complex specification consisting of more than 200 tasks. Besides scheduling, the complexity is linear in the number of tasks and allows handling such complex systems very easily. The produced code is quite readable using well-defined signal names derived from the specification and, as mentioned earlier, the code is synthesizable.

2.3.4 RoseRT to SystemC translation

Another team in our department is exploring a similar method to translate from UML to SystemC.[88] A RoseRT wrapper of SystemC has been built to produce SystemC code from restricted RoseRT design. Despite its intention as a tool for general purpose software development, RoseRT has close similarities to SystemC. Capsules in Rose RT communicate via ports and protocols just as modules communicate via ports and channels in SystemC. A capsule undergoes a state transition when a specified trigger signal arrives whereas in SystemC this corresponds to an incoming signal on

one of the ports specified in the sensitivity list of an SC_METHOD. RT2SystemC translator exploits these similarities to identify and extract important sections of the C++ code generated by the RoseRT tool.

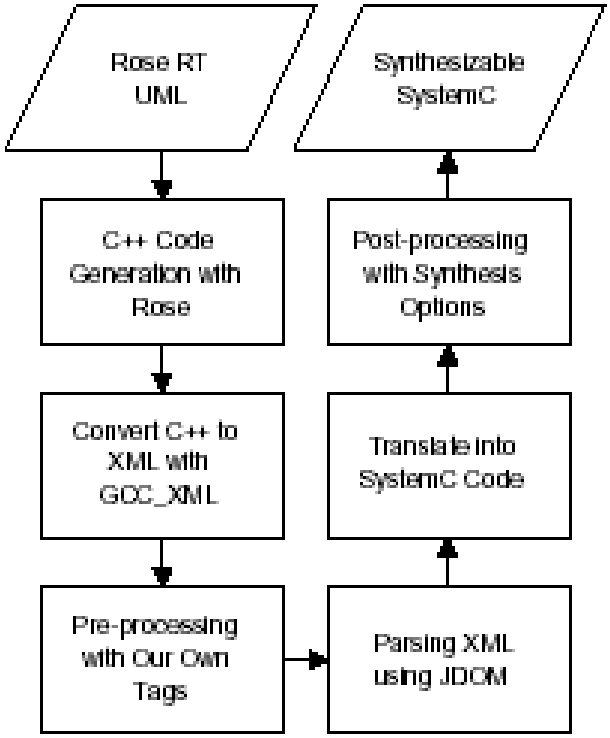


Figure 7: Translation flow of RT2Code

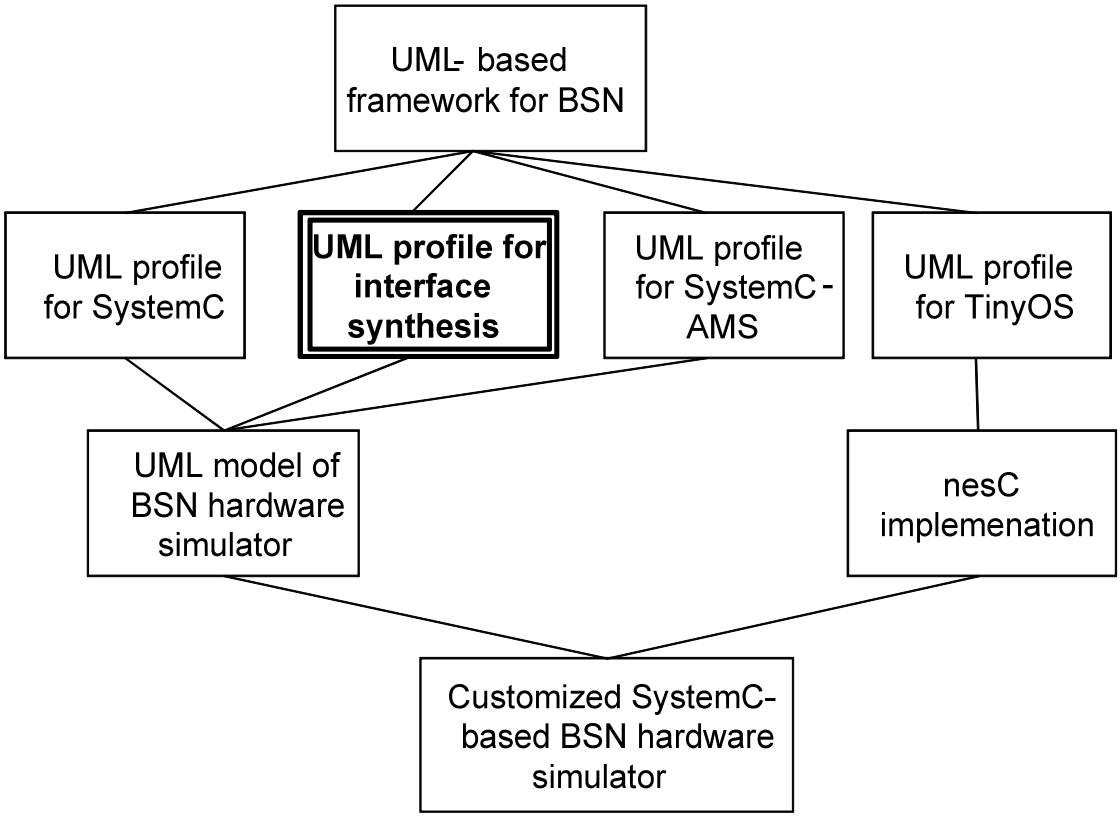
RT2Code translation starts from UML model in RoseRT, and then the rose generated C++ code are further compiled into XML documents. RT2SystemC generator uses the XML documents as input, and generated synthesizable SystemC code. There are interesting similarities as well as differences between their translation and our approach. (Figure 7)

There are still some limitations in the translation. Firstly, the generation is based on the generated code of RoseRT, and it is very software dependent. Furthermore, RoseRT lacks of complex statechart support, therefore, the behavioral functionalities of larger system may not be well captured. By studying their project, we can add in more value to our approach.

2.4 **Summary**

In this chapter, we have presented some background information on UML, SystemC, and they will be the essentials in the framework. We also presented some previous works on UML-based design framework targeting at SystemC implementation including our UML-profile for SystemC.

Chapter 3 Heterogeneous IP Integration based on UML



In last chapter, we presented some background information on UML to SystemC transformation, which can be used to design a functional building block for embedded system. In this chapter, we focus on the customization of existing IP blocks. As the legacy IPs contribute to most of current SoC design usage, maximizing the reuse of them will greatly cut the design effort.

3.1 Contribution of This Chapter

With growth of embedded systems, systems are too large to be handled by single team, and a system has to be partitioned into smaller parts and designed by different engineers. One of the greatest design challenges for hardware developers today is to integrate different parts of hardware system together. IPs, or Intellectual Properties, are predefined functional blocks which have been tested for future integration. The use of pre-designed IP blocks to reduce the complexity of system integration has gained popularity lately. Using pre-designed IP blocks leads to the reduction of time and complexity of system level design. However, these IP blocks often have interfaces that are incompatible due to differences in protocol and/or unmatched I/Os. Integration suffers from these incompatibility issues which also hampers design exploration especially when there are many alternative solutions [65]. The problem of incompatible IP protocols is well-known, and efforts have been made to address it by standardizing the communication protocol. Several standards have been proposed. Among these, the Open Core Protocol (OCP) by OCP-IP [59] has gained wide industrial acceptance. Today, many IPs are OCP-compliant. However, for existing non OCP-compliant IP

cores, it is expensive to customize them to comply with the OCP standard. The process of IP integration also suffers from mis-matched I/O ports. For example, a 16-bit processor will have problems connecting to a 32-bit bus interface. These kinds of situations require logic to be introduced in between the interfaces. Design of such logic circuit is typically done manually and is therefore tedious and error prone. In chapter, we address the above problems by the automatic generation of OCP-compliant adapters for non-compliant components with fixed interfaces from UML structural and behavioral models.

With the shorten time and cost requirement, IP reuse becomes more important than ever. Interface synthesis consists of interface modeling and realization. The incompatible problem has been addressed by previous works in the literature. A common way to capture the system interface specification is the utilization of software programming language or an interface description language, such as variants of C, C++, or Java[32][67][79][94]. Such a specification has the advantage of being executable, and thereby facilitates early verification and simulation. However, for the purpose of system level specifications, the use of these programming languages does not satisfy all the requirements. One key issue is that the different phases of a system design flow — namely the requirement, design, implementation, and deployment — are not sufficiently separated. This can seriously confuse the issues that have to be addressed by each of these phases because of duplication or oversight.

Earlier efforts have also been made in context of timing analysis and verification. Interfaces synthesis tools based on specifying low level data port behaviors through timing diagrams have been proposed. In [21], a method to synthesize interface blocks

that consist of logic circuits and software routines were presented. Other works propose the use of control flow graph [61], event graphs [41], and signal transition graphs [6] to model interface behaviors. All these works address the interface synthesis problem for certain platform environment. However, there can be a large choice of IP. Therefore, the unified modeling for heterogeneous IP would benefit designers working through a top-down design process. UML-based approaches also have been studied [53][66][85]. However, these methods are not matured enough and the mapping rules to low level synthesis are usually too simple to handle realistic scenarios.

Our approach differs from others in the use of high level UML notations for *heterogeneous* IP integration. The following are features that are unique in our approach:

- To ensure correctness and reusability, we use UML structural and state diagrams to specify and formalize system interfaces. This single model is used consistently throughout the entire design process. It not only gives a system level view of the design but also allows for reuse in future designs.
- Automation is applied in every level of abstractions, and between different environments. Code is generated from the same source model, minimizing ambiguity.
- Our framework supports both interface protocol customization and glue logic generation, thereby maximizing IP integration.
- All changes are applied at the higher level, and user will only need to deal with the high level design decisions.

The main goal of this work is to generate the communication links between predefined blocks with minimal user inputs. In chapter, we present our solution which makes use of standards to enable transparency as well as the early validation of designs and the subsequent verification of resultant systems. Figure 8 shows the design flow of our approach. We built UML profiles to capture the system level communication interfaces. The solution is laid out using UML structural diagrams. State diagrams are added to customize functional behaviors of the interfaces. We built a software tool to automatically generate the interface and glue logic to connect the devices while meeting bandwidth and performance requirements. We experimented with our implementation under different scenarios including the "plug-and-play" of OCP-compliant, Verilog and PCI-compliant components into a SystemC simulation environment. The automatic generation of interfaces and glue code leads to the fast prototyping of possible design solutions. These can then be tested and optimized.

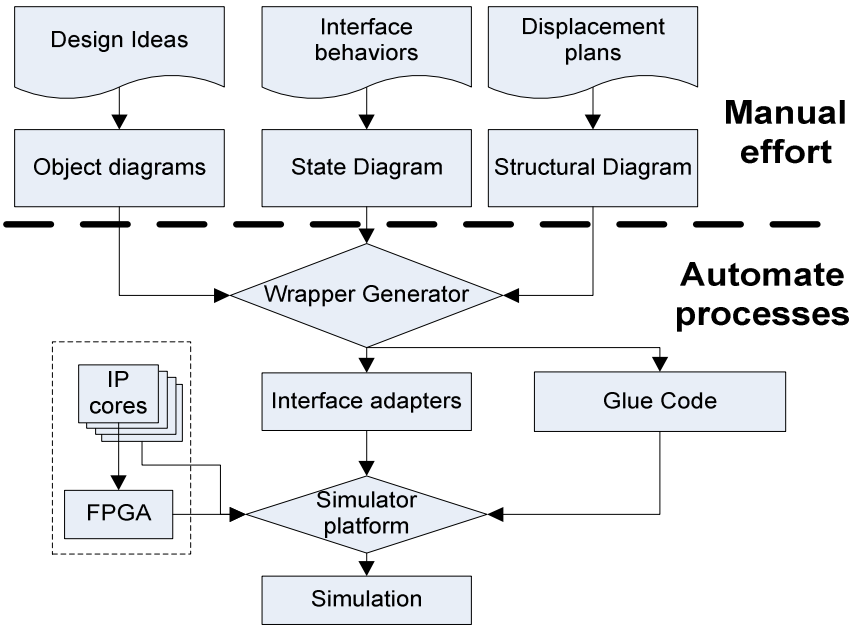


Figure 8: Design flow of UML based interface synthesis

3.2 **Problem Description**

To clearly explain our framework, we define following terms to represent different notations used in our system.

- *UMLport*: UML notation which called *port* in UML level, and it is presented in a small square with UML interface attached.
- *SLport*: System Level data port, and they can carry certain types of data.
- SystemC port: Port defined in SystemC, and they are usually defined as *sc_in*, *sc_out*, or *sc_inout*.
- *Interface*: A point of interaction between system level components, and they are key component in system level designs.
- *UML interface*: A UML notation represents an interaction between UML modules.

We will define our problem scope first using the terminologies. IP blocks can be viewed as black boxes which communicate with each other through pre-defined interfaces. Each interface contains several *system level data ports* (SLport) that can send or receive certain type of data. Consider a scenario where an IP block is connected to an existing interface of a system. Let's assume that the IP block has only one exterior interface. A failure to perform the connection may be due to one of two possible reasons: (1) the interface protocol does not matched, or (2) the SLports of one interface are not sufficient to drive the other. We say that an IP core is *compatible* to an environment if all the output ports of each interface are able to drive the corresponding input port of the other interface while satisfying all timing constraints. For two

compatible interfaces that have different protocols, they form one of two types of compatible pair:

Type 1 compatible pair: There is a one-to-one correspondence between the SLports of the two interfaces.

Type 2 compatible pair: The output SLport of one interface can be made to drive the input SLport of the other through some runtime transformation of its data.

To establish a connection, signal mapping must be setup between the interfaces. For a type 1 compatible pair, a port-to-port mapping is specified through a *contract* associated with the connection. For a type 2 compatible pair, glue logic has to be introduced. We will synthesize both types of the connection from their descriptions specified in *wrapper classes*.

3.3 **User Input**

Graphical notations can give designers a direct view of the overall picture, and it can facilitate designers on adaptation and control of their designs. Unified Modeling Language (UML) from OMG is one of the most successful graphical formalisms[80]. It has been proven very successful and is widely used in software designs. In our framework, we chose UML to be our specification language for its user friendliness and wide adoption. UML provides a large set of notations. We carefully chose a subset for modeling the interface communications. In our framework, we will use structural diagram and statechart diagram to capture the interface models.

UML structural diagrams are predominantly used to describe the component structure of a system. IP blocks and their wrappers are treated as black-boxes and modeled as UML classes. We begin with modeling the communication interfaces of existing IP cores. Wrapper modules with interfaces adapters will then be modeled. After the classes are drawn, interfaces are added as UML ports to capture interface information. We shall use 'UMLport' to distinguish these ports from SLports. Communication channels are modeled as connections between the UML ports. To fix the inconsistency between unmatched interfaces, state charts are added to a wrapper's model to describe the behavior of the glue logic needed to drive the interface. The modeling procedure can be divided into following three steps:

Step 1: Formalize the IP interfaces using UML notations

IP cores are modeled as a UML class with UML ports attached. An UMLport is a real-time system elements introduced in UML 2.0. It is a property of classifiers that specifies a distinct interaction points between the classifier and its environment or between the classifiers. These UMLports will capture the module's communication interfaces. Each UMLport models a group of one or more SLports. The associated SLports will be described in the properties of the UMLport.

The details of an IP's interfaces are captured in the properties and contracts of its wrapper's UMLports. A contract specifies the services that a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment [80]. If two UMLports have a port-to-port match in the communication path then they will share the same interface contract. A connection will be established

between them directly or indirectly (through other ports). The stereotype of an UMLport specifies the communication protocol. In our experiments, the default stereotype for UMLport is SystemC. However, we also used other stereotypes such as OCPSystemC, Verilog, and PCI. The communication parameters of interfaces are captured in the UMLport's attributes. For IPs with interfaces matched to each other, port-to-port communication paths are specified directly in their specification, and only external UMLports to the environment are used for generating the wrappers later on.

Step 2: Define the wrapper classes

To mask the interfaces, the internal UMLports of IP models are connected to the user defined UMLport in the outer class, i.e., a *wrapper class*. The diagrams capture two main aspects of the models: the structure of the model and the characteristics of the interfaces.

The links and blocks in the structure diagram lay out the structure of the model. The wrapped classes communicate with each other using defined channels, and the connections between the internal components are hidden. The adapters are defined using the exterior UMLports of the wrapper classes. We shall call these UMLports *adapter port*s. Each adapter port has a stereotype that indicates the protocol of the outgoing communication. Each wrapper class can hold one or more adapter ports, while each adapter port can have different protocol types and connects to different external modules. Table 1 summarizes the mapping between UML notations and the components of our model.

Figure 9 shows an example of a model after wrapping. The arbiter has 2 clients — one non-blocking and one blocking master. `master_nb` is the UML model of a SystemC module. The UMLports (say `read`) of this UML module represent groups of SLports, of the underlying SystemC module. The interface of `simple_bus` includes the interfaces to the arbiter as well as to the memory, because they have no compatibility problem, we will not model the details of these interfaces. In this example, the simple bus' SLports (and thus the encapsulating UMLports) are implemented in an OCP protocol, while the clients are not. Therefore, two wrapper classes were used to customize the interfaces of the clients. The clients will communicate with `simple_bus` through the OCP adapter ports.

Step 3: Define the behavior for incompatible interfaces

Connections are made between adapter ports and the environment. For connections without port-to-port matchings, additional models have to be added to capture the logical relations between input and output SLports. As an example of a connection that requires glue logic, we consider the IDCT filter used in our MPEG-2 case study (see adapter port ① of Figure 13). To implement the 8X8 IDCT filter, some IP blocks take an address of two dimensional 8X8 array, while other takes 64 inputs in a linear array. To solve this problem, we introduce an *interface state machine* to transform the signals. Our framework allows the user to customize the predefined interface behaviors using such state charts.

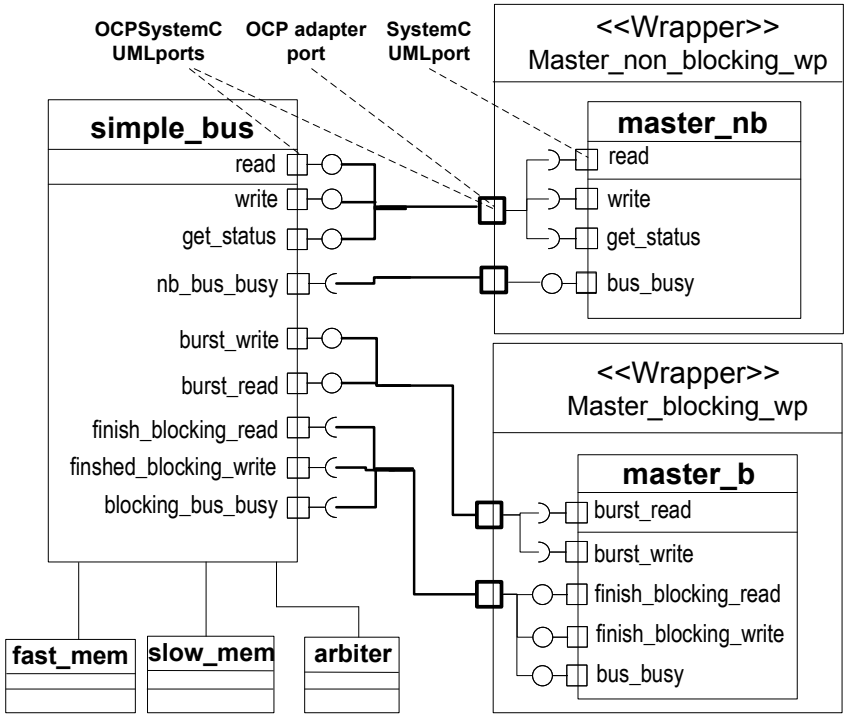


Figure 9: Structural diagram of Simple-bus example

UML Notations	**System Properties**
Parent Class	Wrapper module
Name	Name
UMLport	Interface adapter
Subclasses	IP cores
UMLPort of subclass	Interface
Port name	Name
port type	Type
Interface type	Direction
Stereotype	Protocol
Port properties	Driving signals
State chart	Driving behaviors
Contract Attributes	Signals
Name	Name
Stereotype	type
Tag	Width
UMLport State Chart	Adapter's control code
States and transitions	Finite state machine

Table 1: Mapping between UML notations and design properties

Figure 12 shows the state diagram of the IDCT example, where the master interface has three SLports: `Fast_IDCT_signal`, `Fast_IDCT_addr`, `Fast_IDCT_ack`, and slave interface has 4 SLports: `start`, `datain`, `dataout` and `done`. The diagram defines how the transformation needed for the output SLports to drive the input SLports across the connection.

3.4 Interface Synthesis

A key feature of our framework is the automatic code generation. In our experiments, we use IBM Rhapsody [68] to input the UML diagrams. The model is then fed into our code generator. The analyzer filters out the interfaces and produces abstract models of the design containing the communication specifications. The adapter code is generated from templates using Velocity [86], a template engine that generates code from predefined templates. The Velocity engine merges the code templates and the extracted wrapper models, and performs the final code generation. Figure 10 shows the work flow of our framework. The code generator reads the model and produces the interface synthesis code in 4 steps:

In the first step, models that have the stereotype `Wrapper` are extracted and the corresponding interface adapter code will be generated. For example, to integrate a module in the SystemC environment, each adapter will be generated as a `SC_MODULE`, a primitive type of SystemC. On the other hand, to adapt a FPGA PCI module, an adapter file will be generated which contains the routines for checking and opening communication channels, as well as the communication routines for communicating

with the board. We assume that all the modules without stereotype are normal modules, and they will not be generated in the compilation process. The interfaces extracted from the models are also stored for future references.

In the next step, the code generator will extract and analyze the connections from the model. According to the stereotypes of the UMLports connected to each adapter port, the connections are classified into groups. Currently, we have generators for OCP-SystemC, OCP-Verilog, and OCP-PCI connections. After retrieving the type of each connection, we can now determine if a pair of connections is of Type 1 or Type 2. If a pair of interfaces connecting to the UMLport shares the same contract but has different protocol types, then they are classified as Type 1 incompatible pair. If the connections do not match up with each other but state diagrams were defined to fix the incompatibility, then they are classified as Type 2.

For Type 1 pairs, we build *adapters* (corresponding to the adapter ports) that forward the output source signals to the destination SLports through the desired protocol channel. We shall now explain how this is done for OCP compliant adapters.

An OCP compliant adapter is built to fit the OCP bus fabrics. There are three levels of SystemC OCP communication models: generic, OCP TL1, and OCP TL2. We chose to use OCP TL2 as our wrapper communication model. The advantage of OCP TL2 is that it allows for burst transfers and the communication is faster and more reliable.

When an adapter gets a message from an OCP channel via an OCP port, it first decodes the message, and then forwards the message to the corresponding interface using a SystemC signal. After the internal SystemC component has completed its

operation, a response message will be created and placed in the OCP channel. Correspondingly, when a wrapper gets signaled by an internal component that wants to communicate with the environment, the wrapper will create a message labeled with identifier number of the interface, and puts it on the OCP channel. After the response is received, it is decoded and a reply signal will be forwarded to the component in waiting.

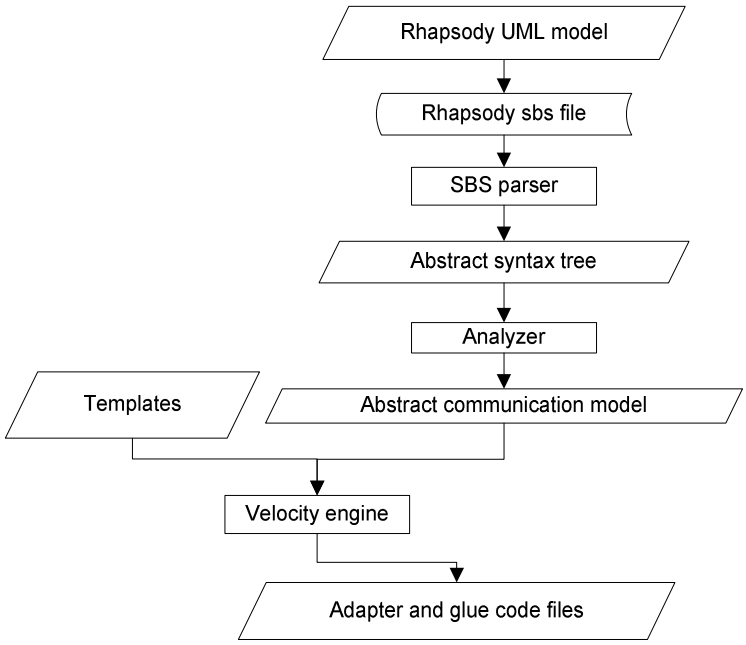


Figure 10: Work flow of wrapper generator

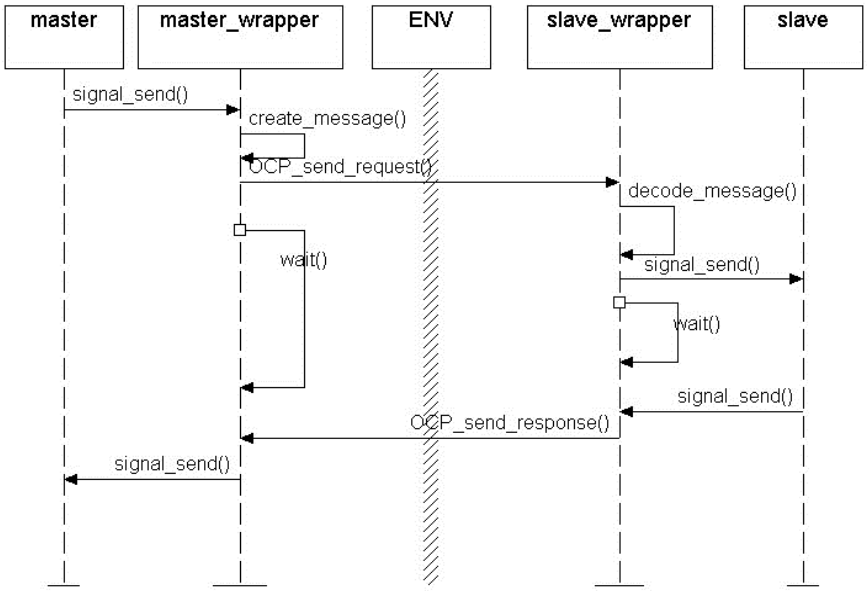


Figure 11: Sequence diagram of OCP communication groups

OCP master and slave ports are predefined OCP interfaces, and OCP communication is performed between them. The pair of master and slave ports forms a communication path. Each communication path starts with one OCP master port and ends with one OCP slave port. A 'provided' interface will be connected to OCP slave port, while a 'required' interface will be connected to an OCP master port. Each OCP port has three threads to control its actions. We call these a *communication group*. Communication groups are paired together and each pair controls an OCP channel. The channel is configured by reading the information extracted from the UML model.

Figure 11 shows the transactions of a communication group, where the *master* is a wrapped component of the master side object, and the *slave* is a wrapped component of slave side object. They communicate with each other using the services provided by

their wrappers, i.e., the communication is controlled by `master_wrapper` and `slave_wrapper` respectively.

Adapters for SystemC-Verilog, and SystemC-PCI are generated in a similar way. A SystemC-Verilog adapter is a SystemC header file that describes the SLport connection of the Verilog module. A SystemC-PCI adapter not only interprets incoming data but also needs to perform the access routines in accordance to the protocol. The wrapping is generic, therefore, multiple levels of wrapping is possible.

For a Type 2 pair, the state diagram shows how the outputs drive the inputs of the client program. A state machine is then generated. Code 1 shows the pseudo code for the state machine that drives IDCT interface using the `Fast_IDCT` interfaces.

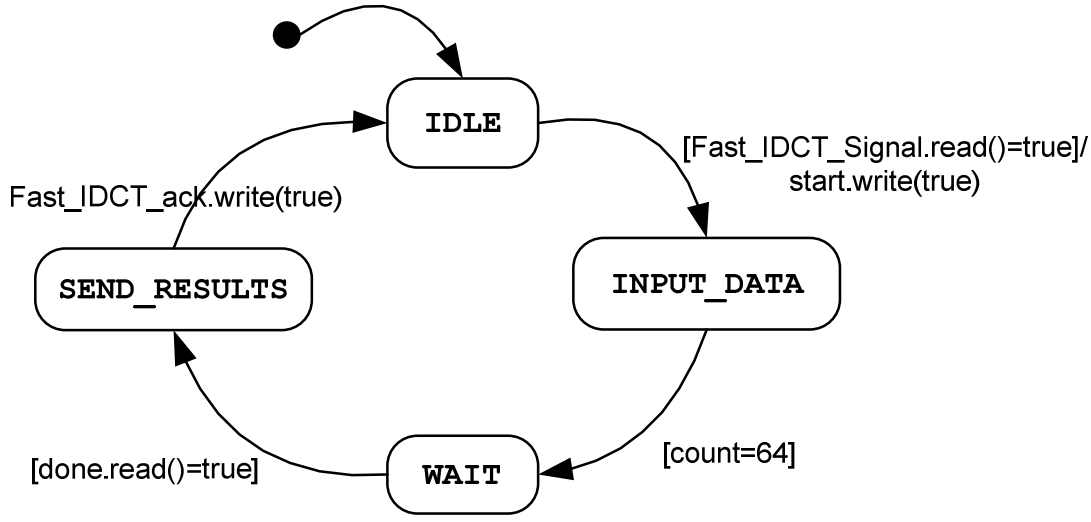


Figure 12: State diagrams of glue logic between IDCT master and slave interfaces at

of Figure 13

```
while(true){
switch (state){
 case IDLE: //in idle state
  wait_until(Fast_IDCT_signal.read()=true);
//state is guarded by Fast_IDCT_signal
   start.write(true); //drive driver port
   state=INPUT_DATA; //change state
 case  INPUT_DATA:
   addr=Fast_IDCT_addr.read();
   for i from 1 to 8, j from 1 to 8
     din.write(addr[j*8+i];
   state=WAIT;
 case WAIT:
   wait_until(done.read()=true);
   state=SEND_RESULTS;
 case SEND_RESULTS:
   for i from 1 to 8, j from 1 to 8
     addr[j*8+i]=dout;
   Fast_IDCT_ack.write(true);
   state=IDLE;
 }}
```

Code 1: Pseudo code of glue logic generated from IDCT of Figure 12

3.5 Experiments and Results

3.5.1 Simple-bus

Our first example, `Simple-bus` is taken from SystemC open source library. We have used it as our illustration example in previous sections. The system consists of a bus kernel, a bus arbiter, two memory slaves (`fast_mem`, and `slow_mem`), and two masters (`master_nb` and master_b). We wrapped the `master_b` and `master_nb` up and substituted the communication channels between the bus and the masters with OCP channels. Figure 9 shows the structural diagram of the `Simple-bus` example after wrapping. We then model the exterior SLports of the bus kernel and the masters,

and wrapped them up. The wrapper code is generated and assembled for the simulation. The unwrapped version of the simple bus SystemC code had 2,684 lines while the wrapped version had 3,743 lines of code. We measured the performance of the unwrapped and wrapped code on a Linux machine with a dual core AMD Opteron 280 CPU running at 2.4GHz. The unwrapped simple bus system finished a million bus transactions in 41.978 seconds, while the wrapped system took 52.188 seconds. Thus, the overall overhead caused by wrapping is 24.3%. Over head is caused by the additional the transactions between wrappers.

3.5.2 MPEG-2 Decoder

MPEG is an encoding and compression system for digital multimedia content defined by the Motion Pictures Expert Group (MPEG) [56]. It is widely used in our audio/video system. MPEG-2 extends the basic MPEG system to provide compression support for TV quality transmission of digital video. We used an open source MPEG-2 decoder originally written in C. We analyzed the functionality and structure of the system, and made it SystemC-compliant.

To test the performance of our adapter code, we customized a SystemC version of MPEG-2 decoder and divided it into five groups. Five wrapper classes were created to wrap up the components. They communicate with each other using OCP channels which are highlighted using bold lines. Each wrapper consists of several OCP ports, and they are connected to other OCP ports as well as to interfaces of wrapped components. The MPEG-2 decoder is partitioned into five communication groups. The

SystemC code is generated and wrapped components are reconnected for simulation using the SystemC simulator.

We experimented with the plug-and-play of the IDCT unit. IDCT (Inverse Discrete Cosine Transform) [17] is a key component in the MPEG-2 decoder system. In our experiments, four IDCT modules were used: F-IDCT and R-IDCT, Verilog-IDCT, and PCI-IDCT. F-IDCT implemented an integer IDCT while R-IDCT used floating point operations. The latter ran at a lower speed but has higher accuracy. They shared the same interface signature but has different interface names. Verilog-IDCT and PCI-IDCT are Verilog version and FPGA version of IDCT respectively. We modeled the IDCT blocks and their interfaces in UML and wrapped them up. `Fast_IDCT` and R-IDCT is a Type 1 pair connection, while Verilog and PCI-IDCT were used to demonstrate a Type 2 pair connection. The wrappers were generated and plugged into the decoder system. The simulation ran correctly even after switching the IDCT module. The rest of design remains unchanged.

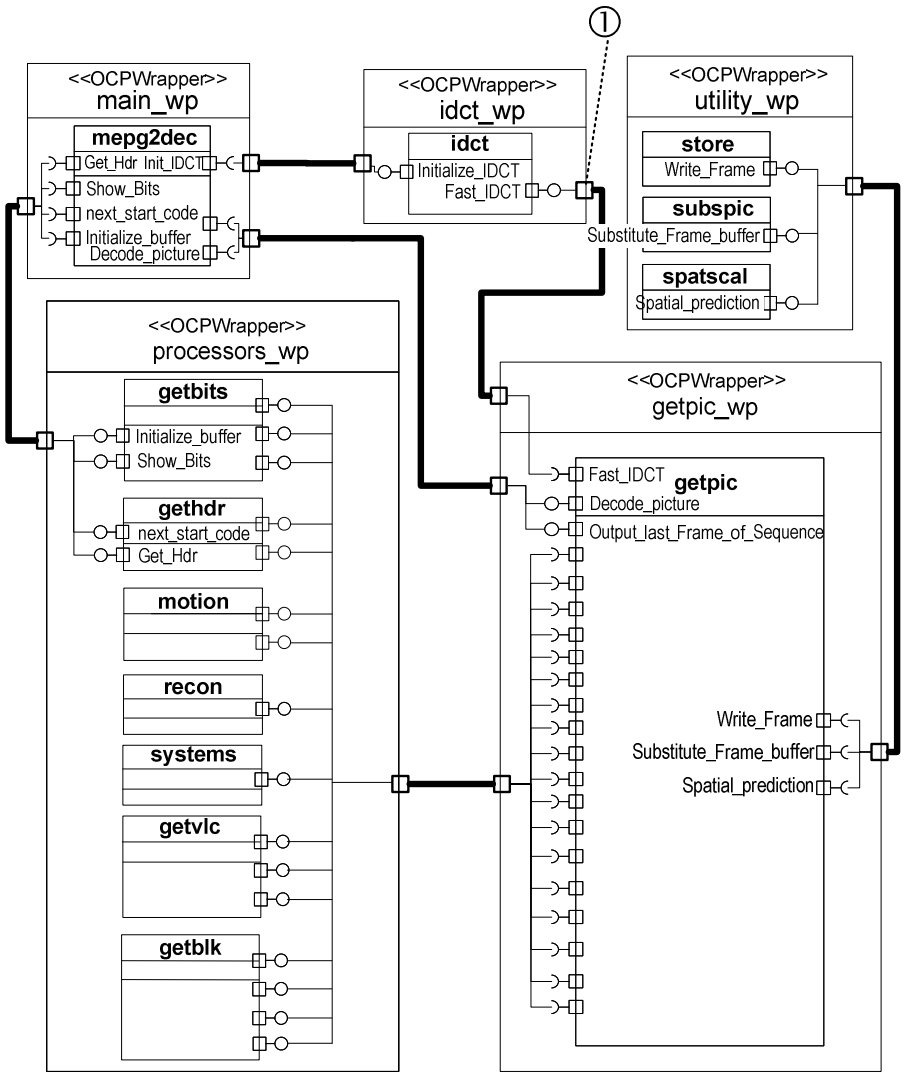


Figure 13: Structural diagram of MPEG-2 decoder after wrapping

Input	Unwrapped w/F-IDCT	Wrapped w/F-IDCT	Extra Overhead
short.m2v	0.412s	0.450s	9.22%
fball.m2v	154.886s	172.415s	11.32%
zoo.m2v	629.155s	730.5275s	16.11%
dhl.m2v	801.406s	932.31s	16.33%

Table 2: Simulation results of decoders with F-IDCT

Input	Unwrapped w/R-IDCT	Wrapped w/R-IDCT	Extra Overhead
short.m2v	0.422s	0.444s	5.21%
fball.m2v	162.84s	184.004s	13.00%
zoo.m2v	699.8335s	789.0175s	12.74%
dhl.m2v	910.759s	1024.9585s	12.54%

Table 3: Simulation results of decoders with R-IDCT

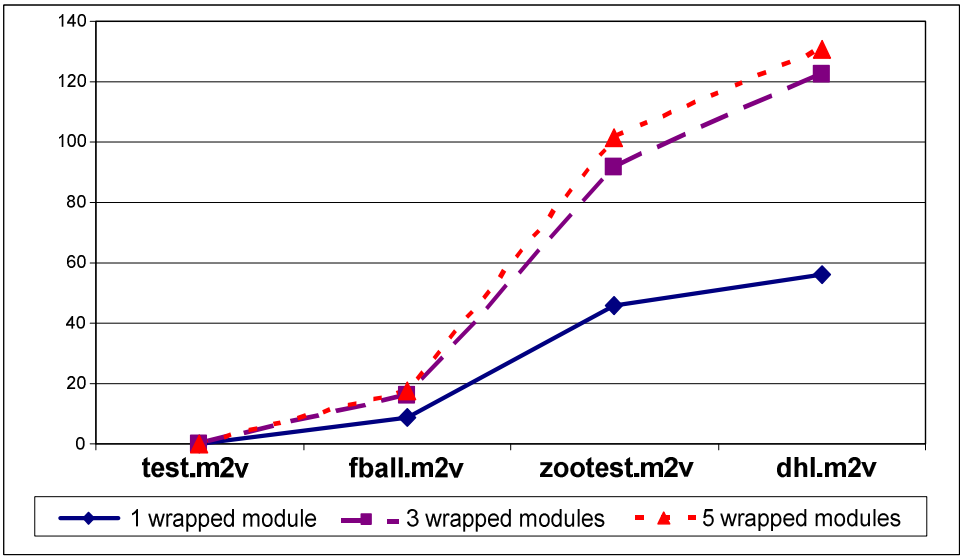


Figure 14: Overhead with different number of wrappers

Because IDCT is the most frequently used component in the system, and has a significant impact on overall performance, the overhead of wrapping has to be measured against unwrapped version of the respective integer or floating point IDCT. Table 2 and Table 3 show the simulation results of MPEG-2 decoder with five wrappers using different version of IDCT processors. From the tables, we can see that the wrapped decoder takes about 9%-16% more simulation time than the unwrapped decoder in both cases.

For Verilog-IDCT and PCI-IDCT, we generated the adapters and then compiled and simulated the entire system using ModelSim. These IDCTs share similar exterior interfaces, and the same UML model can be used for the two cases while marked with different stereotypes. The results of these versions of the MPEG-2 decoder as well as those simulated in the SystemC simulator were identical to the correct outputs of the original program. The simulation is, however, slow due to the overhead of co-simulating in both SystemC and ModelSim.

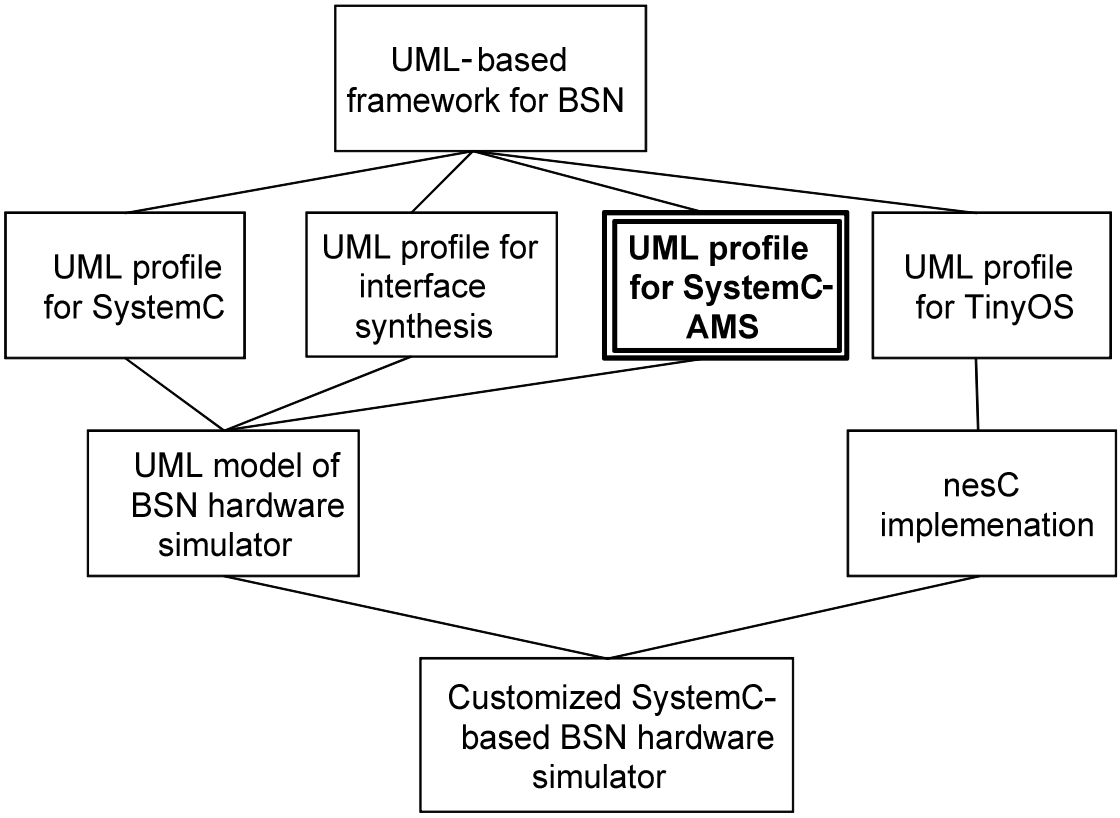
The test result also shows that the overhead of wrapping is not proportional to the number of wrappers (Figure 14). Instead, it is proportional to the number of transactions passing through the wrappers. Wrapping components with lower workload will have a lesser impact on overall performance. There is therefore a tradeoff between configurability and performance. The case studies also show that wrapping can easily be applied to any number of components or groups of components. With the support of our synthesis framework, we can plug different IP implementations into a system, and automatically generate the code needed for system-level simulation.

3.6 Summary

In this chapter, we presented a framework for the integration of heterogeneous and incompatible predefined IP cores. It involves using UML to specify high-level communication models and the automatic generation of glue logics. Two types of incompatible interface are supported: interfaces that are port-to-port matched but use different protocols, and interfaces that require more complex logic to fix the incompatibility. Structural diagrams are used to layout the system, while additional state diagrams are used to model the interactions between interfaces. We tested our algorithms under several environments using SystemC, Verilog, and FPGA modules in a SystemC-OCP environment. The resultant glue logics can be used to test the designs using simulation. Our experiments show that the performance overhead of our wrappers is acceptable.

This algorithm can be extended easily to cross platform designs. For systems that operate in different environments, bridges can be modeled and generated using our framework. Currently, the Velocity templates are hand-written. As a future work, we are exploring the use of behavioral diagrams to automatically generate them.

Chapter 4 Analog and Mixed Signal System



In this chapter, we present the part that how UML-based system level design framework can be applied to mixed signal system design. There have been previous proposals for the use of UML in hardware-software co-design, and we will use it also in the modeling of system consists of both digital and analog components. Our design flow generates SystemC behavioral level code for simulation and verification. We will describe the notations used and how this automatic code generation is performed. We will present three case studies on digital and analog communication components.

With the growth of integration, increasingly more digital chips are designed with analog components embedded. Continuously time-to-market pressures necessitate the raising of the level of abstraction. However, to date it is still not possible to have high level models of a comprehensive system involving software and both analog and digital hardware [10]. As illustrated in Figure 15, a typical embedded system will consist of three portions: software, digital, and analog. Software is normally talked with digital components, and the digital components are connected to analog devices. The Unified Modeling Language (UML)[80] has been used extensively as system level modeling of software. Tools exist that will automatically generate implementations in high level programming languages (C++, Java, etc.) from UML descriptions with support of domain specific models. On the other hand, design of digital side of the system is well supported by CAD tools. However, system level tools for analog modeling tools are not as well developed. Therefore, analog subsystems cannot be modeled at the same level as their software and digital counterparts as shown in Figure 15.

It would be ideal to have a unified framework that can model all components of an embedded system and their interactions. Previous works [37][88] have been demonstrated that UML can be used to represent both software and digital subsystems, and mapping can be established from UML to the appropriate hardware description language for prototyping implementations. If there can be a similar methodology to map analog subsystems from UML to a mixed signal language, then an overall description of mixed signal system can be completed, and designers can model seamless integration between the three portions. This in turn will complete the modeling process, bridging all three partitions of an embedded system's design. There are many advantages to such a rapid prototyping framework. To reduce the time-to-market and cost, it is necessary to evaluate key aspects of a design including the trade-offs involved in meeting timing, performance, and space constraints. Early verification through this prototyping helps to demonstrate and validate design concepts. Rapid prototyping is also helpful in testing various implementation strategies and identifying implementation bottlenecks. Furthermore, a unified description will help clarify requirement, design and implementation specifications.

Unlike digital and software design, analog designs require a high level of analog domain knowledge, especially in mathematics. Matlab [55] provides powerful mathematical support and is widely used as tool for mixed signal design. Mapping Simulink modules to hardware has also been proposed [47]. Our framework will also take advantage of Matlab to incorporate mathematical models in the process of code generation.

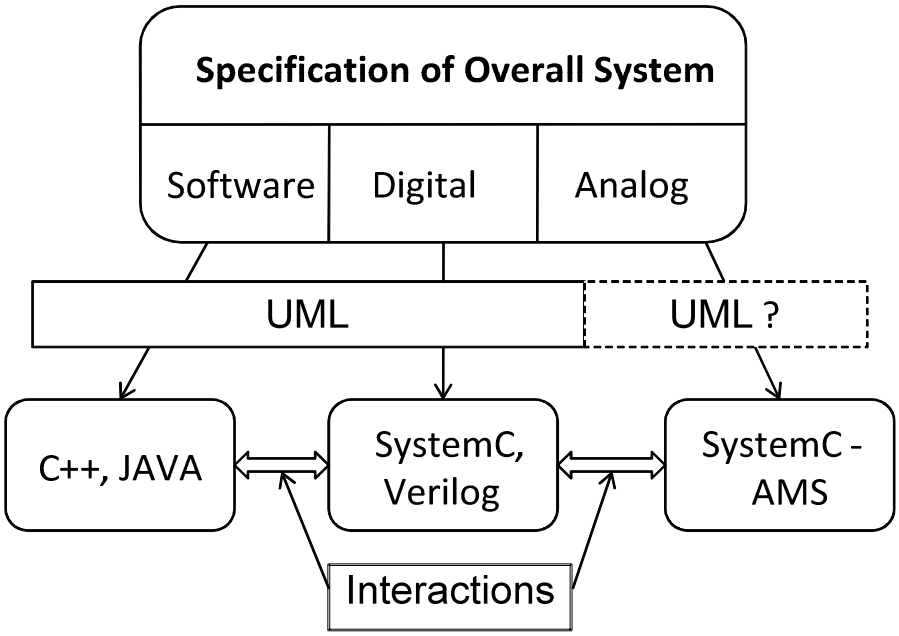


Figure 15: Typical embedded system design and partitions

In this chapter, we will show how UML can be used to effectively model a high level mixed system that includes both digital and analog subsystems. Designers will then be able to capture the overall system specification at UML level for all three partitions of the design. We propose combining the high level abstraction capability of UML 2.0 with SystemC-AMS [76], extending UML to express mixed signal design concept while maintaining the mappings between them. The code for arithmetic computations can also be derived from their Matlab models and embedded into UML models. Both digital and analog components of the system are uniformly modeled using UML-notations. This approach allows for the automatic conversion from unified UML specifications of both digital and analog components to executable SystemC(-AMS) implementation. Our design flow produces not only system level, intuitive, and reusable graphic models for mixed signal designs that also leads to the automatic

generation of the detailed implementation for simulation. Our objective is to build a unified framework to model and generate the topology and interactions for different types of subsystems. We have implemented a code generator to experiment with our ideas. We would like to distinguish our work from those involving hybrid systems[43], where it involved fine granularity interactions between the discrete and continuous elements. Our approach, on the other hand, targets component-level digital and analog entities of coarser granularities. We believe this more common in practice.

4.1 SystemC-AMS Overview

SystemC-AMS[76] is an extension of SystemC[75] to describe mixed-signal design, and it has been popularly studied to model mixed signal systems such as an inertial navigation system [18], the I2C protocol communication [50], and wireless sensor network node [53]. It currently supports two semantic models: conservative and multi-rate synchronous dataflow (SDF) [76]. At the moment, the conservative view is restricted to linear networks and does not allow for the design of real analog subsystems. The SDF modeling formalism used in SystemC-AMS expresses *continuous time* (CT) models as directed connections of dataflow blocks on ports. The multi-rate SDF domain is of particular interests in our experiments. In SDF, the behavior of an analog component is defined as a *cluster* which is a set of interconnected modules with communicating input and output ports. A cluster is managed by a dedicated SystemC process that handles synchronization with the rest of the system. When scheduled by the SystemC simulation kernel, a dataflow cluster runs at a

constant time step, defined by the sampling duration time assigned to one port of one of the modules and automatically propagated to others. The communication between dataflow blocks is represented by streams of sampled signals (discrete in time, discrete or continuous in amplitude), called *SDF signals*.

A SDF module is the basic structural building block for describing CT behavior in SystemC-AMS. Other than the attributes and functions normally found in SystemC modules, there are SDF ports and member functions. Behaviors of SDF modules are defined in SDF member functions. Four types of member functions are currently predefined in SystemC-AMS, they are namely: `sig_proc()`, `sig_post()`, `init()` and `attributes()`. During every time step, `sig_proc()` will be evaluated once, and it will read from input ports, compute and propagate the result data to output ports. `attributes()` may be used for setting port attributes (e.g. sampling rate, delay, sampling period, etc) and `init()` can be used to initialize member data or data ports. `post_proc()` will be evaluated at very end of simulation.

A SDF port is an object that provides a SDF module with a means to communicate with its surroundings. A SDF port has the `sca_sdf_interface` interface and may be a simple port (one interface) or a multiport (N interfaces, N > 1).

There are currently four classes of SDF ports:

- `sca_sdf_in` and `sca_sdf_out` ports are respectively input and output ports that allow SDF modules to communicate.

- `sca_scsdf_in` and `sca_scsdf_out` ports are respectively input and output ports that allow SDF modules to interact with discrete-event SystemC (`sc_signal`)

The number of samples during every time step is set by SDF attributes, Thus, if the rate attribute of an input port is 2 (using `inp.set_rate(2))`, then 2 samples will be read during a evaluation cycle. The evaluation cycle is set by using `set_T`(duration).

4.2 Modeling AMS Design Using UML Notations

UML profile has been used effectively to model SystemC components. In our work, we further extend the profile, by adding domain specific information to the UML modeling elements. We use UML structural and state chart diagrams for modeling. Structural diagrams are predominantly used to describe the components structure of a system, while state chart diagrams are used to capture the module behaviors.

4.2.1 Structural diagram and communication specification

Each module is modeled as one class in structural diagram. To address the needs of abstract representations of mixed signal system in UML, we resort to a few notations introduced in UML 2.0, namely *ports*, and *flow*. Ports and flows provide the means for specifying the exchange of information between system elements at a high level of abstraction. This functionality enables user to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the

system specification evolves, the abstraction can be refined to eventually match that of the concrete implementation.

A one to one mapping is established between module interfaces and UMLports. Each module interface consists of one or several data ports, and each data port carries one type of data. The notation of UMLport is defined to capture the data interaction between the components. *Contracts* are used in defining complicated incoming and outgoing interfaces. Our work is done in the IBM Rhapsody [68] UML 2 context. In Rhapsody's terminology, the interface contract for incoming messages is called a "provided interface", while the interface contract for outgoing messages is termed a "required interface", and they are shown as full or semi cycles attached to UMLports. Each UMLport support at least one contract, and the attributes of contract define the characteristics of the incoming or outgoing data, such as the name and type of the data. All of the components are modeled as a capsule with ports attached. To differentiate the type of signals that the ports can handles, each UMLport is marked using stereotypes, namely *digital* or *analog*.

In mixed signal designs, two types of signals are used for communication. Digital signals consist of pulses or digits with discrete levels of values, and they are non-continuous signals, changing in individual steps. All the signals modeled in SystemC are digital signals (`sc_singal`). Analog signals are continuous signals that vary in time, and they do not have constant value over steps. SystemC-AMS library provide the means to model analog signals (`sca_sdf_signal`). Mixed-signal system design can be viewed as a collection of black boxes with exterior interfaces, communicating with

each other using the proper signals. Generalized system information is captured in UML structure diagrams. Due to the language differences, stereotypes and other extensions are used to differentiate analog communications.

We use class hierarchy to describe the computational entities together with their methods and their attributes. However, class diagrams are also used in a crucial way to give the overview of a system in terms of components and how the components are connected to each other. Flows are drawn between ports, which indicate a message transferring from the start port to the end port. *Flows* link the components together to form the communication. Settings of port are stored in their tags, where tags are pair of messages that can be stored in the description. Four types of tags can be defined in a port's property. They are sample time, rate and delay. We have added some features using UML stereotypes extension mechanism, where digital part and analog part of the system can be distinguished using the defined stereotypes.

Composition classes are supported in our approach, and designers can better manage complexity by grouping several objects together to form nested structural diagrams. .

4.2.2 State chart diagram and behavior specification

Behaviors of components are modeled using state chart diagrams. Each UML classes can have one or more state charts to define the interactions between its UMLports and environments. For digital components, the state charts are translated into finite state machine[37]. The code that implements the states are derived from their specifications or Matlab. However, for analog components, up to 4 states state charts can be used to

define the behaviors of each evaluation steps. Figure 16 shows a complete state chart with 4 states. As mentioned in Section 3, behaviors of mixed signal components are defined in its member functions. We use states with the same name to capture these actions in state diagram. The computation code is directly derived from their Matlab models. It can be tested through Matlab simulation. A state chart of analog component needs to contain at least one state. One mixed signal component may have more than one state charts to define its behaviors of communication with digital and analog components.

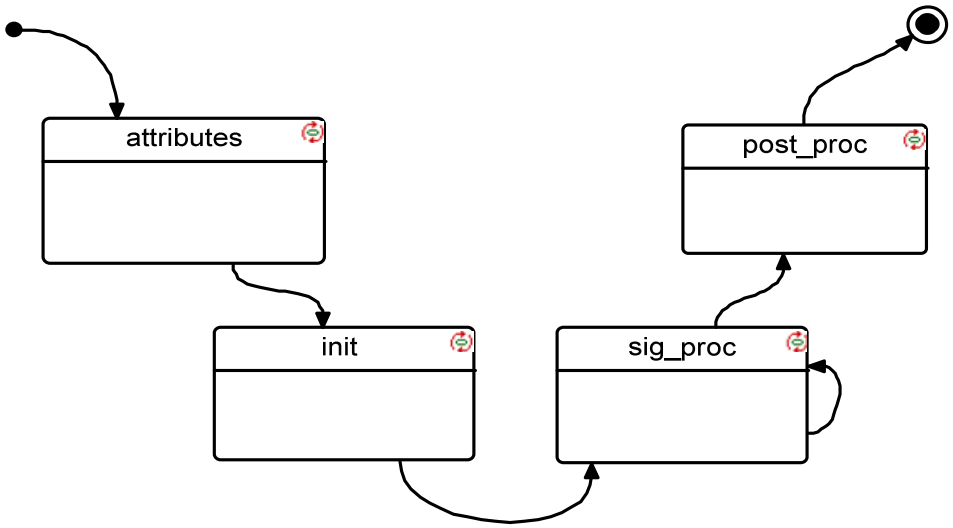


Figure 16: Complete UML state chart of mixed signal components

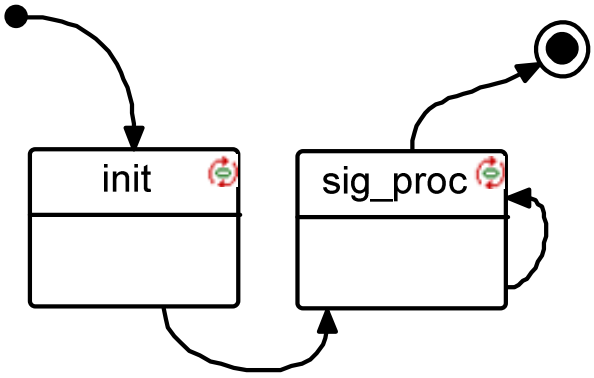


Figure 17: UML Class and state chart diagrams of low pass filter

We will use the low pass filter (LPF) from section 5.1 as a running example. As shown in Figure 17, LPF is modeled as a class with name "*lpf*", it contains several attributes, and a constructor. The state chart will contain only two main states, namely, `init` and `sig_proc`. And the implementation code of member functions will be embedded in the actions of states. Two ports are attached to the lpf class, namely, `inp` and `outp`. They are labeled using stereotype *analog*.

4.3 **Implementation**

As the UML frontend, we used Rhapsody which supports the main features of UML2.0. The intermediate representation is the Rhapsody sbs document file which we parse to gather the necessary information to build the abstract syntax tree(AST). The AST can be used as input to a template engine. SystemC-AMS code is generated using Velocity [86], a template engine which generates code from predefined templates. With the help of the Velocity, the parsing of the modeling document is decoupled from code

generation so the target code can be changed without affecting model extraction. Figure 18 shows the workflow of the code translation.

In our design flow, users do not have to specify the types of the components at the UML level. These will be identified based on the exterior ports that they own. If all ports are digital, then it is a digital component. Similarly, if all ports are analog, then it is an analog module. Otherwise, it is mixed. Each component is mapped to a SystemC module of corresponding types: digital to `sc_module`, while analog and mixed type components are mapped to the SDF class, `sca_sdf_module`. Attributes and primitive operations are written directly to their module declarations.

All the ports and the flows connected to them are extracted from the module. The declaration of the SDF ports are then generated and added to the module's code. Based on the flow directions, the digital ports will produce `sc_in`, `sc_out` or `sc_inout` ports. Analog ports are produced in one of two types - `sca_sdf_in`, or `sca_sdf_out`. To connect between a digital port and analog components, the data port on analog modules has to be defined as `sca_scsdf_in` or `sca_scsdf_out`, and the declaration need to be exported as public. The export code segment will be added to the initialization of the module (normally inside the main class).

Behaviors of modules are generated according to the state charts. For a digital component, a process is created for each state chart, and a FSM is created to manage the interactions between the process and environment. For components with analog portions, the implementation of member functions is extracted from the corresponding state's actions.

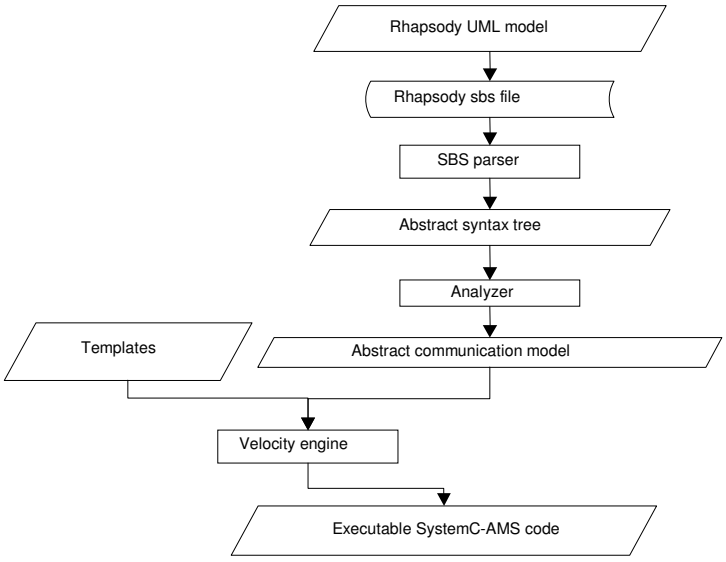


Figure 18: Workflow of our implementation

The top module is generated to initialize the simulation. Constructor of the `top` class contains the code to create and instantiate the object. The connections of the components are also created. `sca_sdf_signal` are created to link the sdf_port together. The top class is associated with every runnable class. It will initialize all the objects, and connect them with the proper signals. Checking for the matching of data types and sampling rates can also be performed. After code generation, validation can be carried out using SystemC simulation. This will enable the early detection of possible design errors, and reduces the costs of debugging.

4.4 Case Studies

4.4.1 Phase Lock Loop

This is an example taken from the SystemC-AMS library. We used it to show how one can model analog circuitry at the UML level and translated the model into SystemC-AMS code. A phase-locked loop or phase lock loop (PLL) is a control system that generates a signal that has a fixed relation to the phase of a "reference" signal. A phase-locked loop circuit responds to both the frequency and the phase of the input signals, automatically raising or lowering the frequency of a controlled oscillator until it matches the reference in both frequency and phase.

The system consists of a phase detector (phc), a low pass filter (lpf), and a voltage-controlled oscillator (vco). A sine wave generator (src_sin) is used as the input, while, after tuning, the wave will be written out for display (Display). The *top* class initialize all the objects of system, in this case, 5 objects comes from 5 classes of modules. Figure 19 shows the class diagram of this example

```
#include "systemc-ams.h"
SCA_SDF_MODULE(lpf) {
//Port List
   sca_sdf_in<double>   inp;
   sca_sdf_out<double>  outp;
//Attibutes List
   double   fp;      // pole frequency
   double   h0;      // DC gain
   double   tau;     // time constant
   double   outn1;   // internal state
   double   tn1;     // t(n-1)
//Operations List
  void init() {
     tau = 1.0/(2.0*M_PI*fp);
```

```
  }
void sig_proc() {
  double tn=sc_time_stamp().to_seconds();
  double dt = tn - tn1;
  outn1=(outn1*tau+h0*inp.read()*dt)/(tau +dt);
  tn1 = tn;
  outp.write(outn1);
  }
SCA_CTOR(lpf) {//constructor
   outn1 = 0.0; tn1 = 0.0;
   fp = 112e3;
   h0 = 1.0;
 }};
```

Code 2: Code Generated for low pass filter in SystemC-AMS

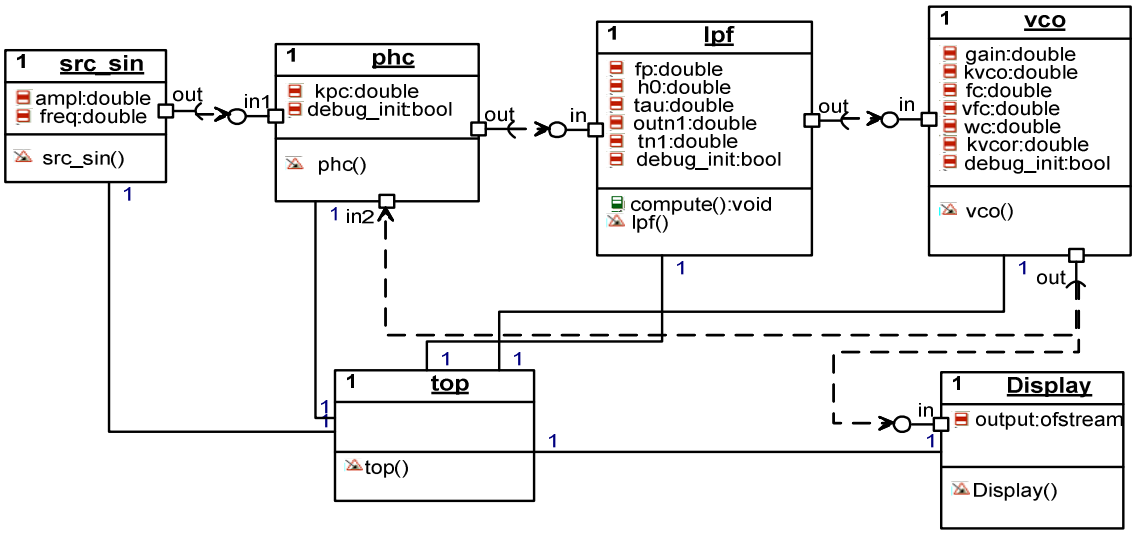


Figure 19: UML structural diagram of PLL example

We first captured this UML-level model using Rhapsody by applying the UML profile. Code 2 shows the generated code for Low pass filter in SystemC-AMS. We then simulate the resultant code using the SystemC simulation kernel. The *sin_src* generate a stream of 7MHz sine wave, and the basic time step duration of analog components is set to 0.001µs. The simulation of generated code execute in same speed as original code from SystemC-AMS library.

4.4.2 Binary phase shift keying transmitter with noising

Our second example is a BPSK transceiver. BPSK (Binary Phase Shift Keying)[64] is a modulation technique that has proven to be effective for low and medium FER operation, as well as for amateur HF work.[64] As shown in Figure 20, the system consists of a transmitter, an antennas/propagation channel, and a receiver. Binary shift keying uses binary polar signals to modulate the phase of carrier by 180 degrees. Digital data is converted to an analog signal by passing through a 1-bit D/A converter. The analog signal is then up-converted, amplified, and transmitted into the noisy channel. The received signal is amplified, down-converted, low-pass filtered, and converted back into the digital domain.

The structure of BPSK System is shown in Figure 20. We use a random generator to generate a sequence of bits as input. The source data's voltage levels are then shifted by *regulator*, and the shifted data is then multiplied with a cosine source to form the BPSK-modulated signal. The modulator blocks are implemented in an ideal manner with no noise. Additional White Gaussian Noise(AWGN) is added to the transmission media. The noise is generated every evaluation cycle with a random value. The propagation channel adds noises to the transmitting signals and propagates them to the receiver. A low pass filter is used at receiver end to filter out the noises. The filtered signals are then fed into A/D converter, and the ADC will produce the demodulated data. The UML-Diagram of BPSK transmitter is shown in Figure 21, the *top* module is waived for clearer view.

The BPSK system consists of several blocks which we modeled using UML(as shown in Figure 21). The model is then fed into translator which generates the corresponding SystemC code. Note that, the digital to analog converter (DAC), and analog-to-digital converter (ADC) are mixed signal components, and the rest of the system are in RF domain. The sampling rate is set to 10GHZ, and the data transfer rate is of 10MBPS. The power amplifier and low noise amplifier each has a gain of 1.0.

Figure 22 shows the waveforms generated from the BPSK simulation. Waveform (a) shows the transmitted digital data stream(taken from the output of Digital/Analog converter). Waveform (b) shows the digital shows filtered signals. The bit error rate (BER) for our simulator is Eb/N0= 0dB. However, when we increase the level of noise, we did pick up some error bits during the transmission.

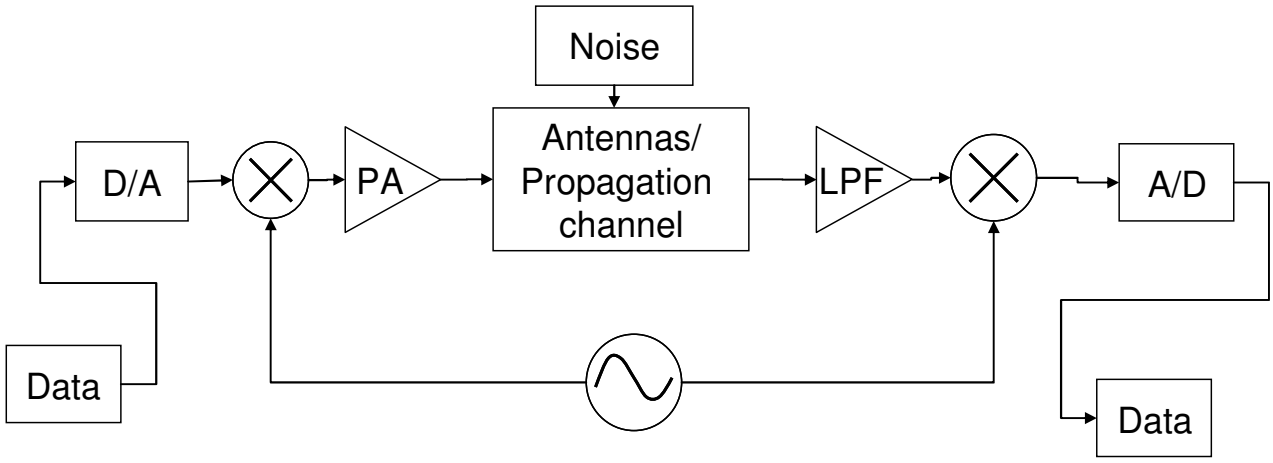


Figure 20: Block diagram of BPSK example

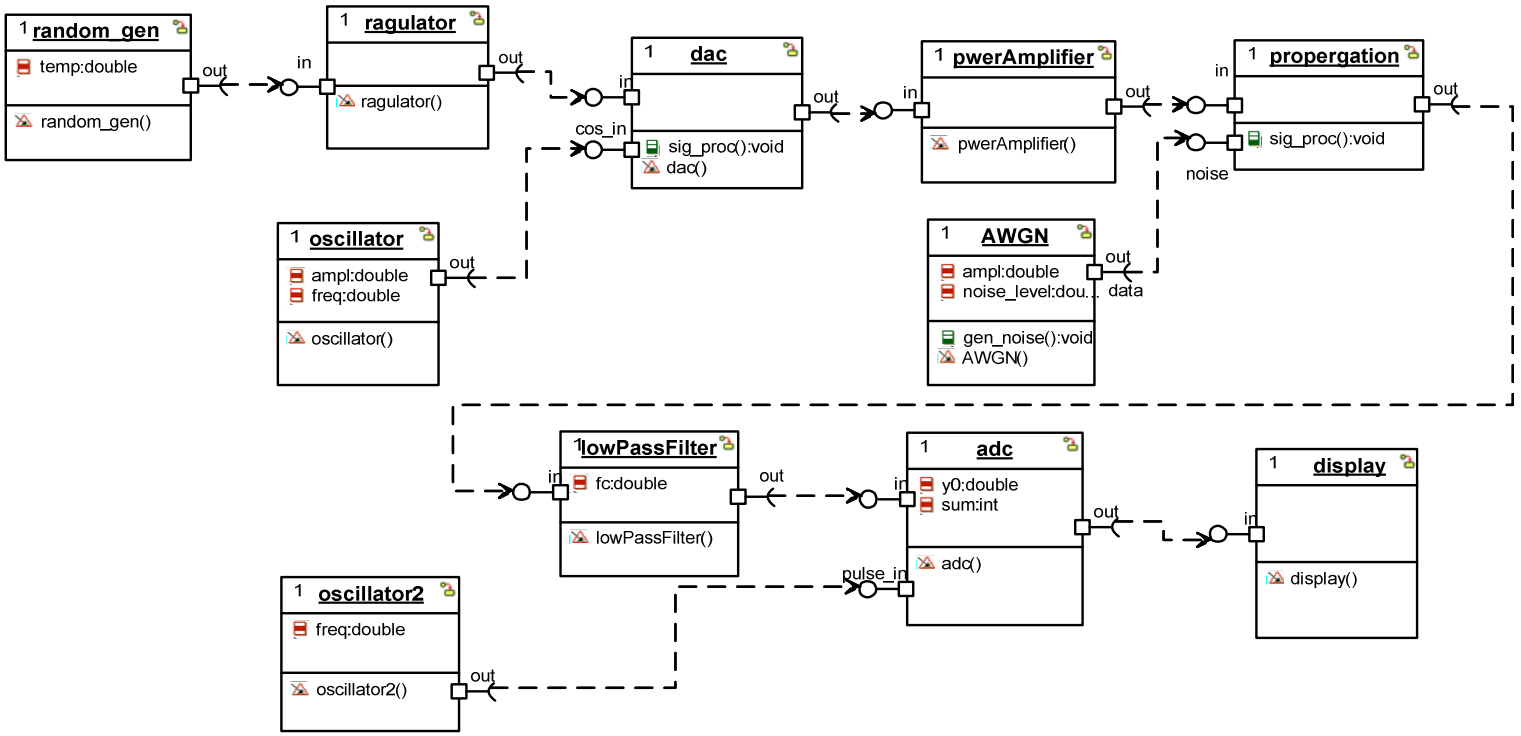


Figure 21:UML Structural Diagram of BPSK transceiver

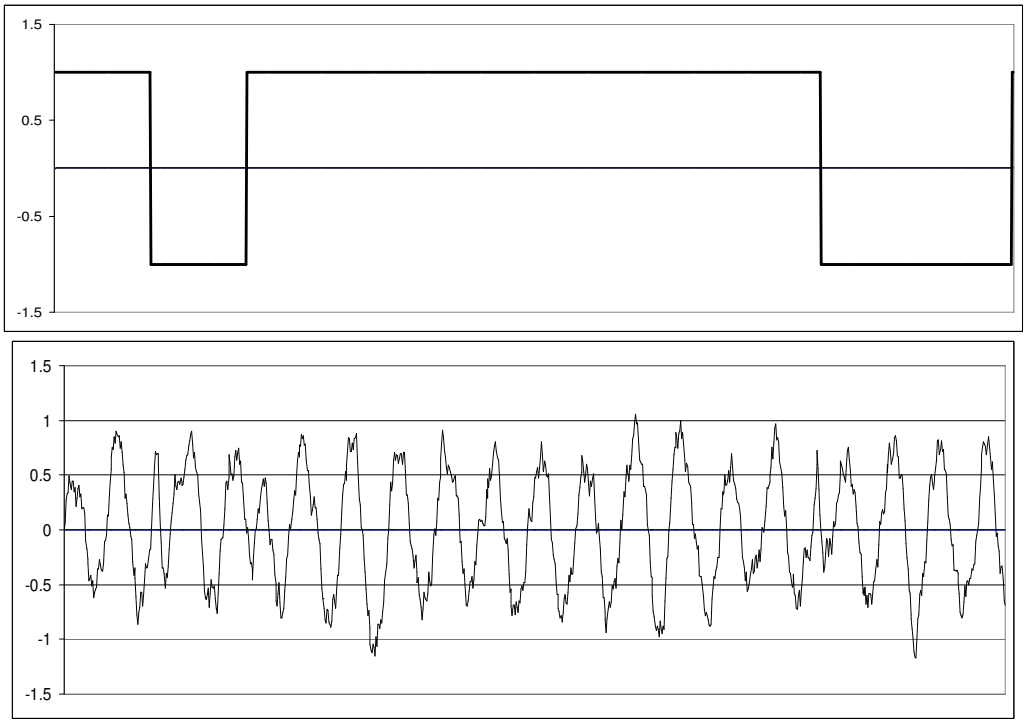


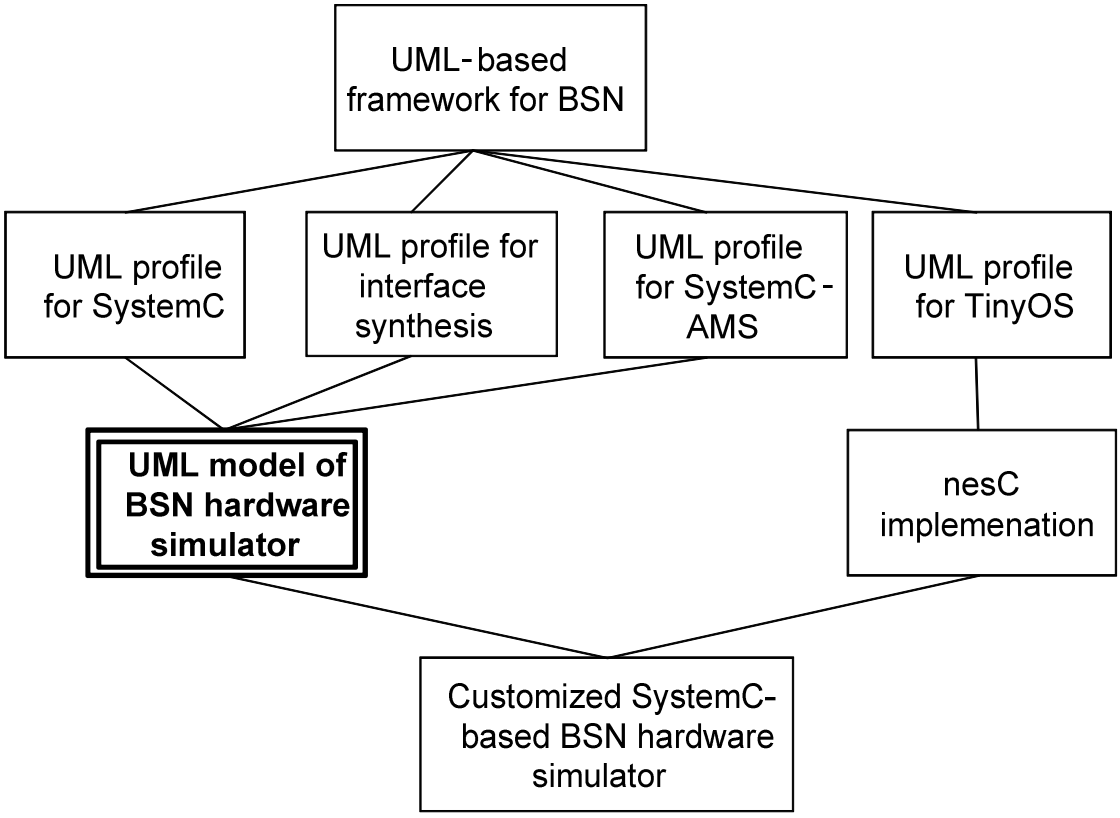
Figure 22: (a) Transmitted data stream (b) Samples after filtering

4.5 **Summary**

In chapter, we have presented a UML-based system level framework that holistically integrates software, digital hardware and analog components into a single design. Our tool automatically translates UML models to SystemC-AMS implementations of analog and mixed signal components. By using ports and flows as well as their stereotype extensions, we are able to capture the interactions between the subsystems and integrate models of communication of analog modules into digital and software models. We demonstrate our approach with several examples, including a couple of analog transmission units.

In the future, we would also like to target our modeling method for hybrid mixed-signal hardware platforms such as ASIC and FPGA chips, especially since the generated SystemC-AMS code is nearly synthesizable. The performance of generated code needs to be improved.

Chapter 5 SystemC-based BSN Hardware Platform Simulator



In previous chapters, we have presented UML profiles for SystemC and SystemC-AMS designs. In this chapter, we will show how the profiles are used to define a SystemC-based BSN hardware simulator. This simulation-based based performance evaluation is going to be the fundamental component for the co-design framework.

The simulator is part of Embedded and Hybrid System II program in Singapore, centered on the development of a Body Sensor Node (BSN) system and of relevant health-care applications.[35][36][77] The project is carried out by Singapore Agency of Science Technology and Research (A*STAR) with collaborations from Institute of Infocomm Research (I2R), Institute of Microelectronics (IME), Nanyang Technological University (NTU) and National University of Singapore (NUS). The simulator provides a high level executable model of wireless body sensor network platform. The model is based on SystemC, an IEEE standard language for system level design of embedded systems. The simulator provides detailed breakdown of the timing and energy behavior of individual software and hardware components in the system. For example, the workload on the sensors can be tuned according to their energy consumption characteristics so as to obtain optimal system efficiency. The simulation speed is fast thanks to the modeling level and techniques that we use to optimize simulation speed. The simulator has been tested with a medical application that monitors the patient wellbeing by measuring the ECG and the SpO2 blood level [19].

As member of the development team of simulator, I was able to understand the detail behaviors of the simulator. Each components of the simulator was studied and modeled using our proposed UML profiles. As a result, we have built a UML-based

simulator model for BSN hardware node. After then, the simulator are no longer the SystemC code, they become visualized functional components. This helps designers to quickly and easily customize the simulator and make it as close to the target BSN hardware platform as possible. Customizations can be applied at these UML models, and the changes are reflected to lower level implementation via our code generator. The UML-based simulator model provides an efficient and re-usable framework for designers to prototype and verifies their BSN hardware designs

In next section, we will present some previous works on BSN simulators. The introduction of our BSN simulator will be divided into 2 parts. In section 5.2, we focus on the SystemC-based implementation of the simulator. We will show the structure of the simulator and how we can use it to optimize an application. In section 5.3, we will present UML-modeled simulator after we have applied the UML profiles.

5.1 **Previous Works on BSN Simulators**

There are numerous simulation tools available to aid programmers in understanding the performance and behaviors of Body Sensor Networks (BSNs). These tools vary widely in scalability, accuracy and feedback details. Surprisingly, relatively few exists for the purpose of timing and power analysis—the two most essential aspects in the design and optimization of body sensor applications. In fact, to the best of our knowledge, none of the existing instruction-level simulation tools for MSP430 platform supports timing and power analysis of sensor motes at the functional level.[36]

Well-known discrete event-based simulation environments such as NS-2 [39], TOSSIM [63], and OMNeT++ [4] provide effective ways to validate the behaviors of network protocols. However, they do not capture internal operations of the individual motes that might assist developers in debugging and optimizing applications.

There have been a number of instruction level mote simulators; for example, Atemu [52], Avrora [71], COOJA [20] and their extensions. Atemu simulates the operations of individual motes and communication between them, though it does not supply timing and power consumption information of the motes. Similar to our simulator, Avora and its extension AEON [58] allow for the evaluation of energy consumption and lifetime prediction of sensor network. However, they are both limited to Mica2 2009 Body Sensor Networks platform, which is not applicable to our context where our focus is on the MSP430 platform.

Eriksson et al. [31] introduces an instruction level simulator called MSPsim, targeting the MSP430 microcontroller that contains a sensor board simulator as well which simulates hardware peripherals such as sensors, communication ports, LEDs, and sound devices. Although comprehensive in features and easy to be integrated into the cross-level simulation plat- form COOJA, MSPsim shares the same limitation as Atemu: it supports source-level stepping and run-time variable inspection only, without displaying any timing or power consumption information of the various components of a mote.

Recently, TOSSIM has been extended to estimate the power consumption of the Mica2 sensor mote. The extension, Power- TOSSIM [51], is built on top of TinyOS and

is highly scalable. This benefit comes at a price as the tool is coupled with the TinyOS written code. In contrast, our simulator works on machine code level, and it can simulate sensor network applications written in any language with any operating system components, such as both TinyOS and SOS. Further, Power- TOSSIM follows a high-level of abstraction approach, which might not provide enough details necessary for application optimization and may potentially lead to poor accuracy in the analysis results.

Several other tools that work at the abstract level, for example, SensorSim [71] and SENS [73], unfortunately, assume rather simplistic power usage and battery models which may not be realistic in actual hardware and practical applications.

In the Embedded System domain, several power simulation tools for energy profiling have been proposed (see e.g. [2][78]). However, most of these tools are limited to profiling of microprocessor energy consumption only as they are designed for general embedded systems.

Despite simulation being the de-facto standard tool for the evaluation of body sensor networks, lately there is a growing interest in the formal method community. Owing to their expressiveness, automata-based techniques have been used to analyze wireless communication and protocols (see e.g. [1], [51]). Recently, Timed-automata has been employed to validate QoS properties of BSNs such as packet end-to-end delay, packet delivery ratio and network connectivity [74].

Another contrasting line of work is the Sensor Network Calculus (SNC) [30], a worst-case analysis framework that uses algebraic techniques. It is developed based on

the Network Calculus in Computer Networks domain. SNC has been continuously adapted and extended to effectively model and analyze worst-case behaviors of sensor networks, such as these examples in [46] and [29]. Although comparatively much less explored and limited by their ability to scale up, formal techniques are much faster than simulation and provide formal guarantees. Thus, they can be used in early development, for instance to identify worst-case scenarios of the systems for a given architecture. With further investigation and appropriately incorporated with simulation-based techniques, such formal techniques will certainly benefit system designers and developers to a great extent.

5.2 SystemC-based BSN hardware simulator

5.2.1 Simulator

We implemented a fast, cycle-accurate simulator for BSN applications. This simulator allows the developer to determine accurately the processing and energy performance of individual modules in the application, under different configurations. The simulator assumes the applications utilize multiple motes which are connected via a wireless BSN. Following is a brief description of the simulator; for a full description please refer to [35].

The mote simulator is implemented in SystemC and it takes in any application code written in NesC. In order to handle a network of sensor motes, multiple instances of these simulators are created as different threads in the same simulation process.

Figure 24 shows the general structure of our mote simulator. It consists of four main components: a micro-controller module, a ChipCon2420 module, a sensor module, and a power monitor. The micro-controller in the BSN IC mote is Texas Instruments MSP430. This module incorporates a CPU module, a clock module, RAM, Flash and other peripherals. The CPU module includes an instruction set simulator and an interrupt manager.

The CC2420 module is an abstraction of the real radio chip. It provides only the basic functionalities, as our focus is to capture information on timing and power consumption.

The power monitor is not part of the MSP430 architecture and it is designated solely to monitor the power consumed by each component of the mote. It is also able to compute the energy consumed by each function of the application being simulated. This information is very useful for determining the functions that need to be optimized in order to reduce the energy consumption.

5.2.2 Application

The SystemC-based BSN simulator can provide simulation for the full system of a generic wireless BSN. Typically, a wireless BSN consists of several wearable sensors on a human body. These sensor motes transmit vital body parameters such as ECG or SpO2 blood level through wireless technology to a gateway mote which in turn is connected to a PDA, whose job is to process data, send commands to the motes, and forward data to a doctor's clinic. Additional information on full system simulation can

be found in our previous work [36]. The users interact with our simulator through a friendly Graphical User Interface. To simulate an application, the users follow these steps:

- Choose the number of sensor motes.
- Choose the application running on the gateway mote and on each sensor mote.
- Start the simulation.

While running, the GUI displays the PDAs screen and a multi-tab window for the motes. For example, the PDA screen can show graphs of data received from the motes. Each tab in the sensor window shows the current status of the corresponding mote. • Stop the simulation. A summary of total energy and timing for each mote is showed on its corresponding window tab.

5.2.3 Functionalities

Our simulator provides the following functionalities to help developers debug or optimize their applications:

1) Total Energy and Timing: After simulating an application, the total energy and timing are shown. This in- formation is extremely important to evaluate the overall performance of an application.

2) Functions Energy and Timing: A breakdown of timing and energy for each application function running on each mote is also listed. These numbers enable developers to optimize the application by providing deep insights into the

behaviors of the application and help them focus effectively on the most time-consuming or energy- consuming functions.

3) Monitoring points: The user can choose several critical points in a program as monitoring points. During simulation, whenever the execution reaches these monitoring points, an instant snapshot of the following information is automatically saved in a log file:

- Energy and timing: This is the energy and time consumed so far by that mote. It gives user a quick look at the rate the code running on the mote is consuming power.

- Register values: The values of CPUs registers are shown for functional debugging. For example, a jump instruction whose destination is indexed mode jumps to different locations depending on the run- time value of the indexing register. Thus, the value of that indexing register could tell which instruction and/or function the simulator is going to execute next. This knowledge is useful for user to validate the control flow.

- Radio buffer: The number of filled bytes in CC2420s transmitting and receiving buffers are captured and displayed. This helps detecting buffer overflow and dropped messages, which enable the developer to adjust the message sizes and the sampling rates.

- Transmission summary: the snapshot provides a count of bytes sent or received so far by the radio chip and the serial port. These data, together with timing information, can be used to compute the average transmission rate.

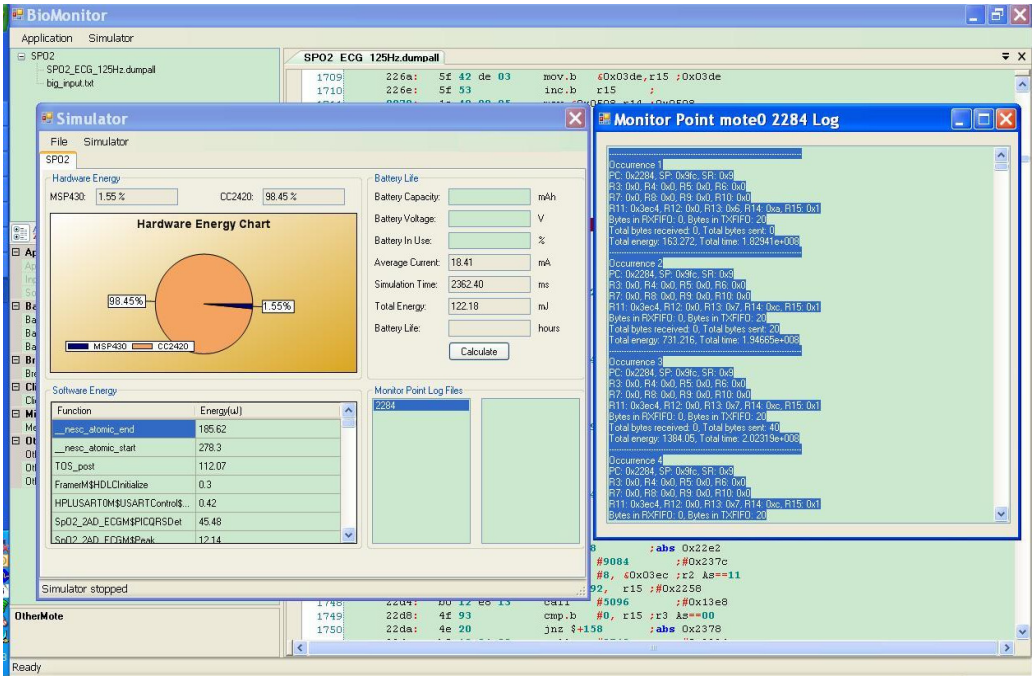


Figure 23: Simulation data collected at monitor points.

Figure 23 shows an example of the collected data at various monitor points. This is particularly useful when the user wants to compute time or energy for a fragment of code which could be part of a function or a combination of several continuous functions.

5) Monitoring memory addresses: The user can choose several important memory addresses in a program to monitor their evolution. These addresses may refer to variables used by the application or they may correspond to the internal registers of various peripherals. During simulation, whenever the value of such a

monitored address is modified the new value and the address of the current instruction are logged to a file.

5) Battery life computation: We employ Rakhmatov's battery model [14] to capture the battery performance and to compute its life time. Since BSN applications are on human body, they require mobility, flexibility and long duration. Thus, the battery life time is a critical factor while developing the applications; the longer the battery can last, the more convenient it is for device-wearing patients.

6) Control flow: Last but not least, our simulator provides users with an option to record the control flow of the application running on a mote in a log file. Since applications written in NesC code are compiled with TinyOS library to generate complete C code and then executable code on a specific platform, details of an 11 applications control flow is only available at C/assembly code level which is significantly different and much more comprehensive than the original nesC code. Understanding the control flow of an application is vital for debugging and optimizing purposes.

5.2.4 Guideline to debug/optimize an application

We give a general list of guidelines which the developers can adopt in the process of debugging or optimizing applications. These guidelines are created based on our experiences and on the general principle of top-down problem solving. The steps and

their order are flexible and could be modified to suit different applications, problems and objectives.

Step 1) First, the user should set up and run the application once to get a summary of the total time and energy consumed by the application for a standard sensor input. Together with a breakdown of timing and energy for each function, this gives a comprehensive overview of the application. Based on this overview, the user might have a general idea if a problem exists and what the problem is.

Step 2) Other information such as battery life and transmission rate could also be a good source to detect the problem.

Step 3) Look at the generated C code or control flow log file to understand the application, such as how different functions are linked together either as caller-callee relationship or as a sequence of tasks/events.

Step 4) Check the control flow log file if everything is in order. If something goes wrong, the control flow may show some unexpected runtime behavior.

Step 5) With a good understanding of the application control flow, the user can identify critical points in the program where local information may reveal the cause of the problem. She can run the whole simulation once more to collect data at these monitoring points. Then she may investigate the result to narrow down the problem.

Step 6) If the problem is with application code, fix or improve it. If the problem is with TinyOS code, consult TinyOS developer guide to find if there is any solution

or possible optimization. After fixing the code, go back to step 1) run the simulation one more time to get the new performance summary. If the result is not satisfying, repeat the steps for further optimization.

This SystemC-based simulator is very fast, incurring only 5–20 times slower than native execution on real motes yet able to provide critical performance data for developers to tune their applications. This is apparently an advantage compared to other methods. For example, running applications multiple times directly on the real hardware gives very little insight about the applications running on it. In chapter 6, we will present an example of how we use our simulator to obtain useful data about the performance of an application and how the developer uses this information to identify and remove a bottleneck in the application.

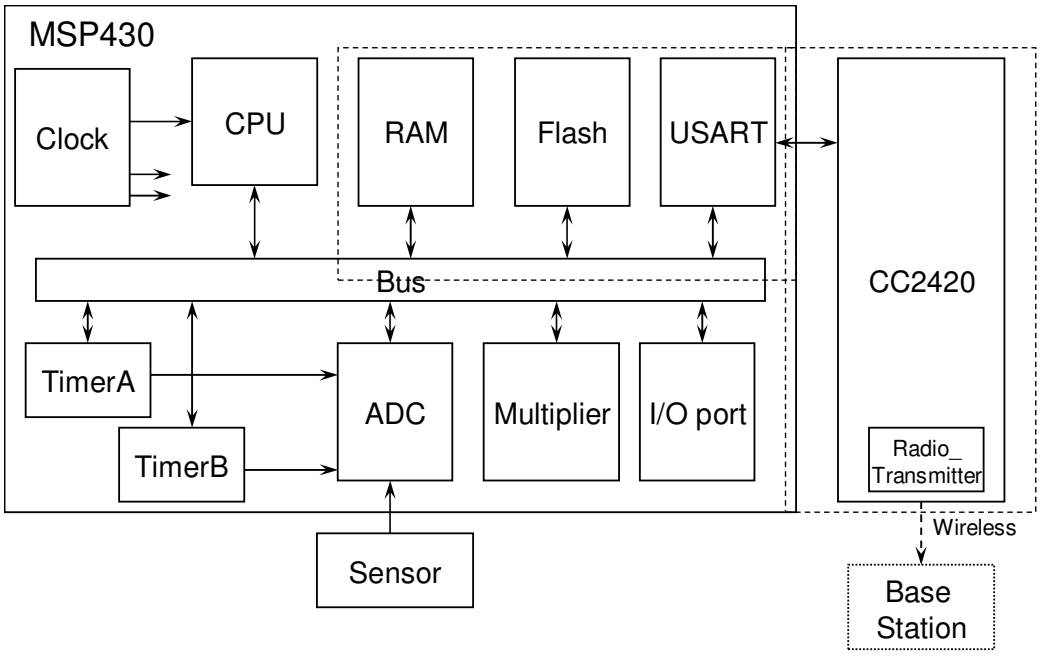


Figure 24: Block diagram of BSN simulator

5.3 UML-modeled BSN hardware simulator

5.3.1 UML-modeled BSN simulator

With help of the UML profiles for SystemC and SystemC-AMS, we constructed UML models for BSN simulator. The structural diagram of this simulator is shown in Figure 27. The UML diagram mainly shows the interfaces and communications between the hardware components, other details such as local operations and properties are hidden for better presentation.

The UML-modeled simulator has two main parts: the digital part process the data collected from the sensors and pass the results to wireless transmitter, where sensors and the transmitter is from analog domain. The RF processing starts from radio transmitter of CC2420, it modulates the data packet and send the data out to base station over the air. Figure 25 shows the selected portion of BSN simulator from dashed area of Figure 24. Note that all the connections in the diagram are from digital domain, except the *transmit* function to the radio transmitter.

The CC2420 uses an OQPSK (Offset Quadrature Phase-Shift Keying) modulator to modulate the data, which is from RF domain. OQPSK is a special version of QPSK(Quadrature Phase-Shift Keying) in which the transmitted signal has no amplitude modulation. UML profile for SystemC-AMS is applied here. Figure 26 shows the composite class diagram of the OQPSK transmitter, in the system, the data packet is formed in CC2420, and then sent to transmitter to modulate and send via digital signals.

The power monitor module *PowerMonitor* is not part of the hardware peripherals; however, we modeled such module to collect energy consumption of selected component at execution time in BSN mote. As shown in Figure 27, CPU(MSP430) and wireless transmitter(CC2420) is under monitoring, and they have interfaces connected to the power monitor module for the state update. Every change in operating modes in these two components will update the power monitor. The power monitor, in turn, will obtain the simulation time spent in the previous operating mode. Power monitor contains the energy profile for the monitored components, which can be obtained from datasheet or experiment. With simulation time, operation modes and the energy profile, power monitor can compute the energy consumed during each simulation period. By accumulating the energy consumed by the individual components and by the BSN mote, when the application completes execution, we obtain a breakdown of total energy.

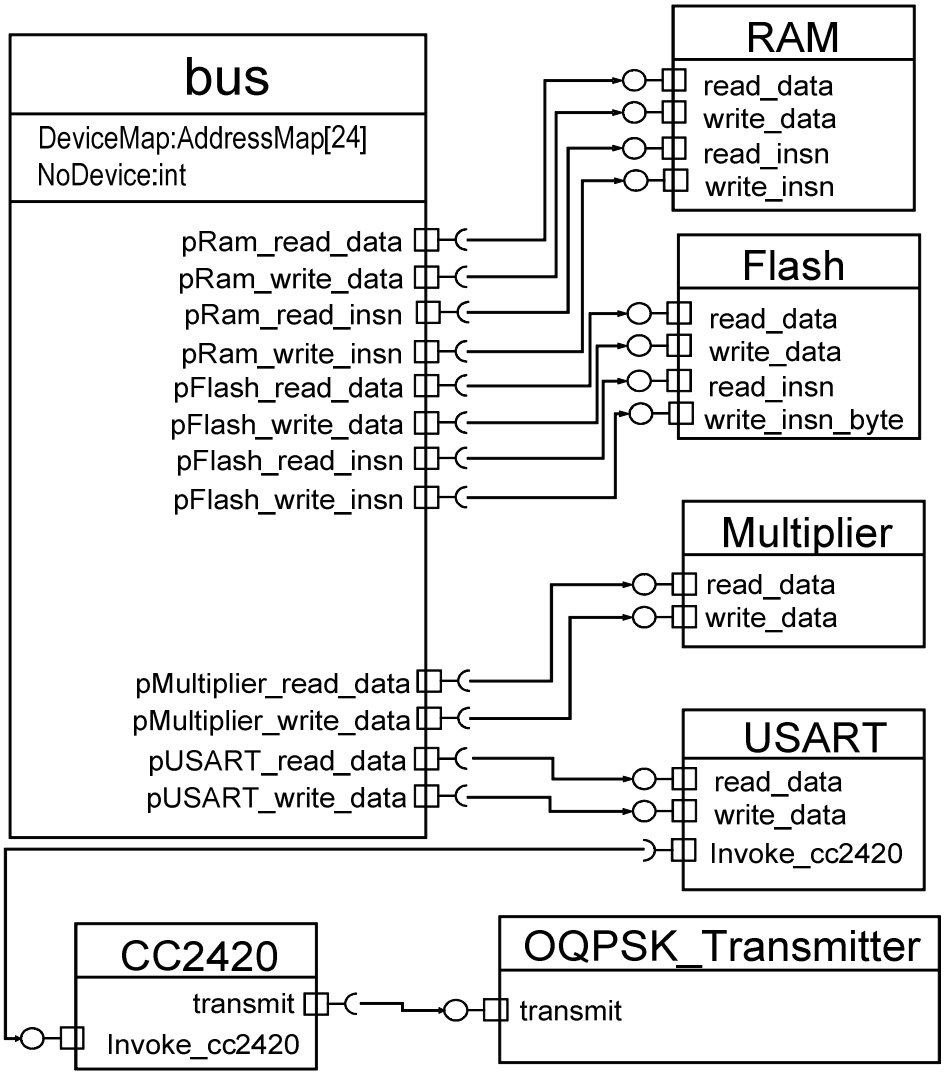


Figure 25: Selected part of UML class diagram of the BSN simulator

5.3.2 Simulator customization

The UML-based simulator can be customized to adapt the hardware changes. For example, in the simulator, MSP430 (CPU) and transmitter are connected to a clock component, and designer wants to increase the clocking rate to achieve faster computation speed and transmission speed. This can be achieved by changing the parameter value that controls the self-ticking rate of *ClockModule* in the UML model.

On the other hand, energy profile of the components is required to be updated accordingly. Therefore, by changing the parameters of clock module and power monitor module in the UML, we can obtain a customized BSN simulator, while other components in model remain unchanged.

Customization of modeled components can be made at UML level. With the external interface fixed(UML-port), the functional specification can be change through changing the embedded code or the state chart attached to this module. For example, if we want to change the functionality of ADC and make perform some filter function, we need to add the filtering function implementation to its class diagram and state chart. The modification will be reflected to generated code.

Modeled components can be replaced with third party pre-defined modules, ie. IPs. We achieve this by applying the wrapping technique outlined in chapter 3. The wrapper ensures that the external interfaces to other component on BSN simulator remain unchanged. After code generation, the wrapper code will be generated, and it will be used to interact with the wrapping component and communicate with other BSN simulator component.

Mapping between the UML models and SystemC implementation has been established through our code generator. The code generation process is similar to what we have presented in section 4. User can customize the simulation environment by changing or replacing the components. The updated simulator can be generated automatically from customized UML model. Key performance issues, such as timing and energy consumption, can now be tested by simulating the generated

implementation on this automatically generated simulator. This highly-reconfigurable simulator provides a fast and accurate performance evaluation tool to aid both software and hardware design.

5.4 **Summary**

In this chapter, we first outlined our SystemC-based BSN simulator. The simulator is a well-defined tool to test and evaluate given BSN software code. The simulator ensures accurate and fast simulation speed. Users are able to evaluation the software/hardware design by doing simulations using the simulator. A guideline was given to optimize a BSN application based on the simulation and energy information collected through simulation.

We further extend the simulation by formalized it with UML profiles that we have defined. The simulator becomes more manageable and reusable through these UML-models. With this high-level UML model, designers will be able to customize BSN hardware in short time, and the auto-generator ensures an executable simulator will be ready as soon as the high level models are refined. The usages of simulation in our framework will be discussed in next chapter.

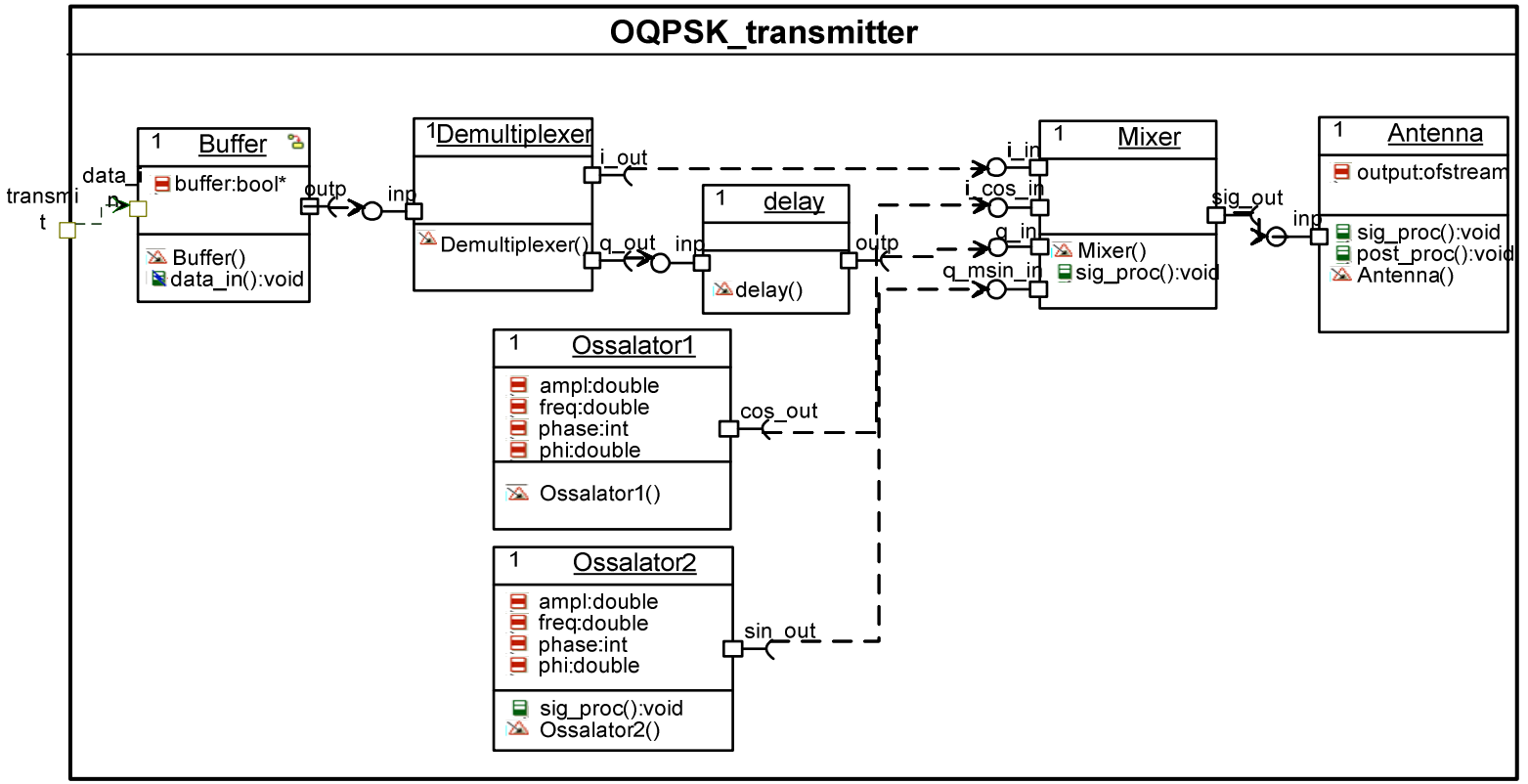


Figure 26: UML class diagram of OQPSK transmitter

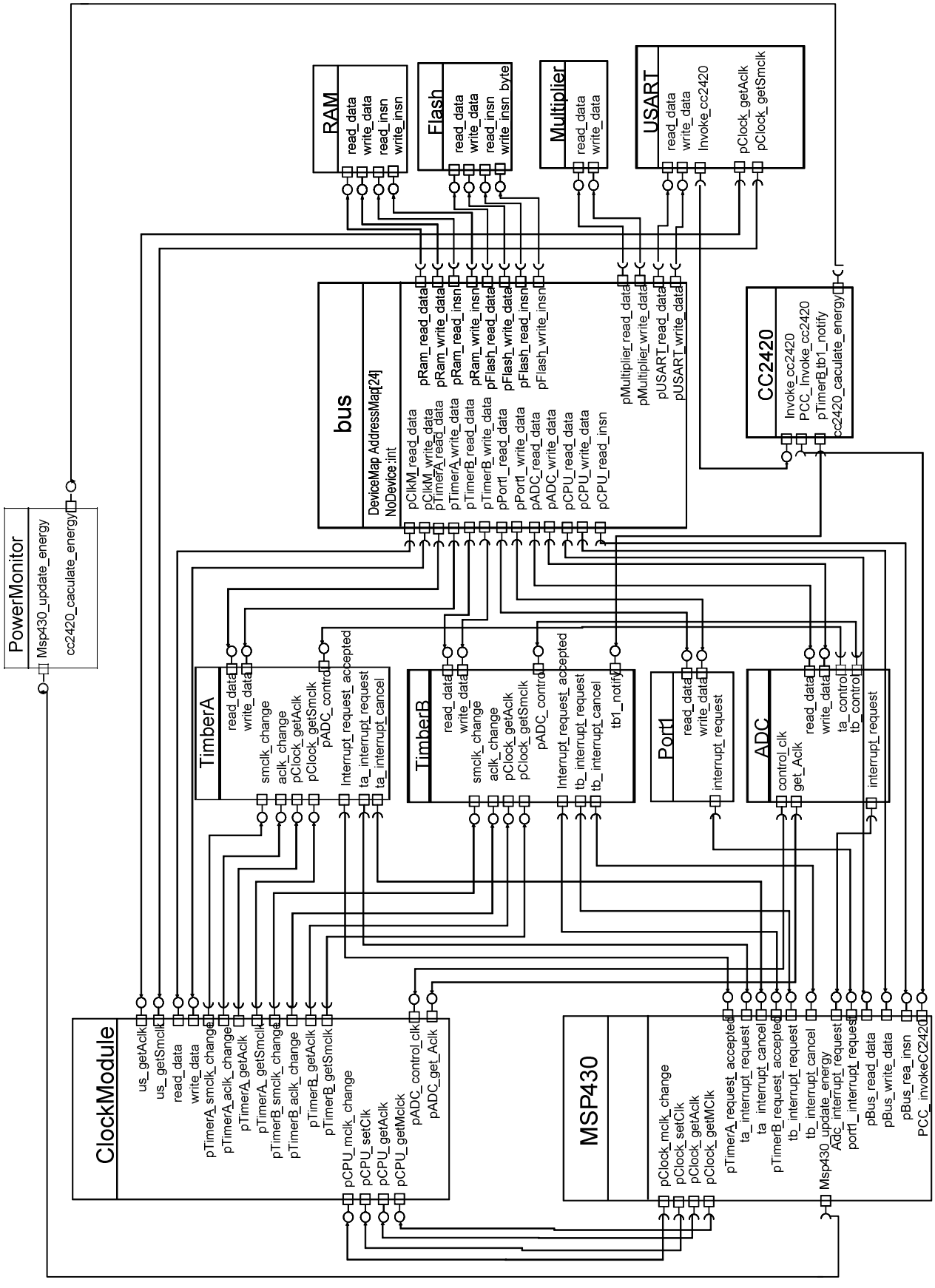
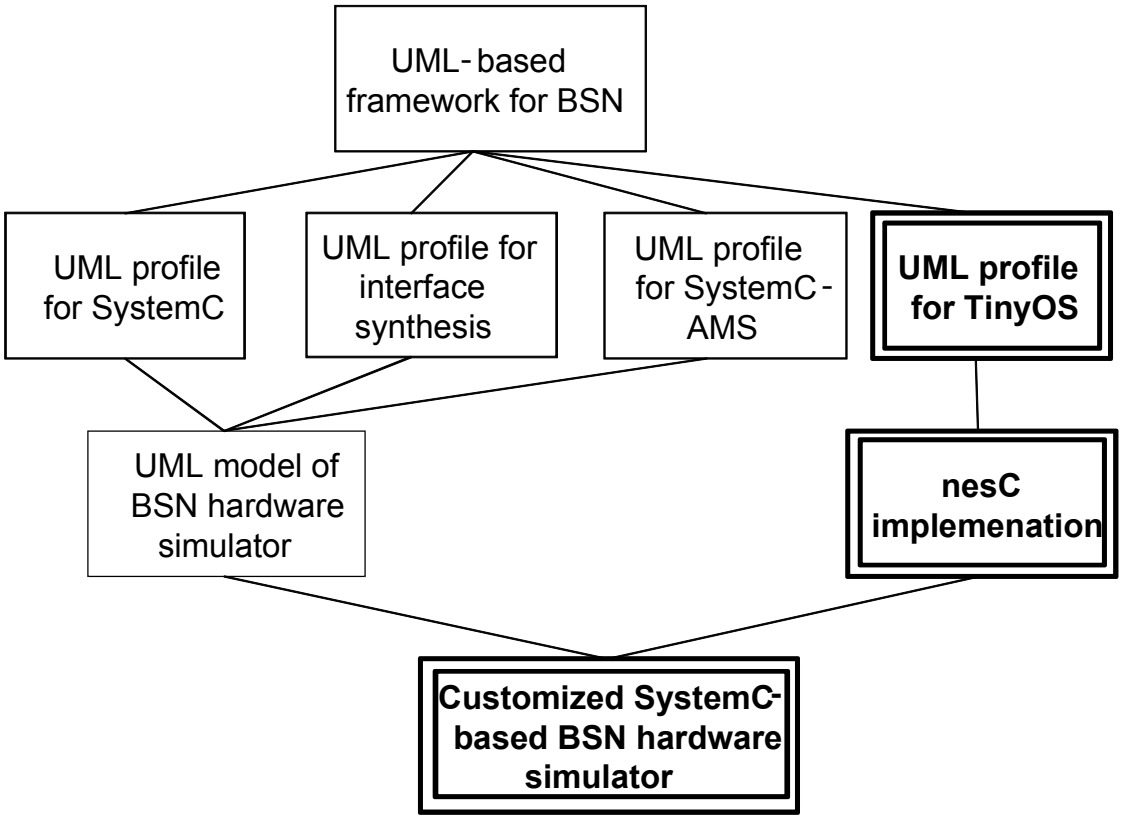


Figure 27: UML structural diagram (interfaces and communications) of BSN simulator

Chapter 6 UML 2.0-based Co-Design Framework for Body Sensor Network Application



In last chapters, we have presented the UML-based design approach for hardware simulator. This chapter proposes a new model-driven development framework, which is intended to manage the complexity of BSN application development. The framework consists of (1) a Unified Modeling Language (UML) profile to capture the specifications of BSN applications, and (2) a fast simulation environment to perform validation and testing.

A *UML profile* [81] extends (or specializes) the standard UML elements (e.g., class and association) in order to precisely describe domain-specific or application-specific concepts. The proposed UML profile abstracts away the low-level details of TinyOS and provides higher abstractions for application developers (even non-programmers) to graphically design and maintain their applications. It also allows developers to understand and communicate their application designs in a visual and intuitive manner. A model repository is designated to store available models of tinyOS components, and it helps designer to create and manage their designs while maximizing the model reusability.

Our UML-based simulator provides a fast and accurate simulation environment that can be customized to capture and simulate hardware changes. Code generator is implemented to maintain the link between simulator UML models and its implementation. The simulation environment is therefore can be customized at UML level. The implementation generated from the software designs can be executed directly on the customized environment. Fast and acute simulation helps user to estimate timing properties as well as energy consumption throughout the software design process.

Both design and verification environment are now unified under the name of UML, and they yield a complete design flow for pre-integration design and validation. Refinement and tuning will benefit from the formalized reuse model. Our design flow essentially supports a "design-generate-simulate-refine" cycle. We shall present several case studies to show how BSN applications are produced from specifications and how refinement can be done effectively with aid of simulation results.

6.1 Previous Works on UML Profile on TinyOS Simulator

UML-based design flows for embedded system designs have gained popularity in the recent years[82]. Several profiles have been proposed to address various design issues [37][45][81]. In our framework, we proposed and implement UML profile for TinyOS-based BSN systems.

Similar design flows based on other languages have also been proposed. In [11], a mapping from high level Specification and Description Language (SDL) models to TinyOS implementations was presented. A driver model based design approach was also proposed [69]. Code generation are outlined or proposed in these works. However, they did not tackle the issues of heterogeneous design. Furthermore, reuse is not well supported in these frameworks. Other researches focus applied UML for wireless sensor network(WSN) designs. In [27], the authors focused on the sensor's behavior instead of the components that reside on the sensor node. A simulation framework for WSN based on VisualSense, a Ptolemy II-based visual language, that captures node

behaviors was proposed in [60]. However, it does not support code generation which we deem as the key to rapid prototyping.

Several simulators, such as OMNET++ [3], J-Sim [38], Em* [44], have been built to analyze runtime behaviors of TinyOS applications. However, to date, none of them have supports the cycle-accurate simulation of BSN node devices. One simulator that has modeled both the CPU and the peripherals of sensor nodes for cycle accurate simulation is the Avrora simulator [5]. However, for the moment, Avrora does not support the architecture that we are targeting, namely a BSN board built with the MSP430 microcontroller. None of these simulators are integrated into a high-level design flow such as ours.

VIPTOS [15] bears some similarity with our framework in that a high level design methodology is combined with a simulation environment. However, the simulation environment of VIPTOS focuses on network level, while our work focuses on hardware simulation. Furthermore, it is based on the Ptolemy flow.

Gratis [26] provides a graphical design environment to create and generate TinyOS implementation. It has some powerful features such as the ability to extract graphical models from nesC code. Their graphical notation, called the Generic Modeling Environment, is based on the UML class diagram and the Object Constraint Language. However, the main aim of Gratis is model the *structure* of the TinyOS application. It therefore does not model the *behavior* of the application. Our framework captures both structural as well as behavioral specifications at the UML level. Furthermore, Gratis does not support simulation environment.

One unique feature has exclusively differentiated our work from others, that is, both design and simulation layers of design flows are now in high reusable manner to cope with ever-changing nature of BSN applications. We believe this will cut down the design and verification cost greatly.

6.2 TinyOS and nesC

TinyOS [12] is a component-based operating system designed to meet programming requirement of embedded networks with limited resources. The components of TinyOS interact through well defined interfaces. TinyOS software is written in nesC [12], a dialect of C. nesC programs are written in the way of interacting components that are connected together by interfaces. There are two types of components in nesC implementation: modules and configuration. Modules implement specific functionality, while configurations define how components are connected.

Interfaces are bidirectional. They consist of commands that are calls made by a user to a service provider, and events which are calls that a provider can make on a user. For example, sending a packet is a command, while receiving a packet is an event. All the interactions between components happen through interfaces. This allows for implementations to be changed easily. For example, a component named App that uses SendMsg can be directly connected to different types of communication protocols without changing App’s code. This "plug and play" style of interchangeability is predicated on how modular the implementations are. In general, the object-oriented

nature of modules with interface encapsulation allows us to reuse the same modules for different applications.

nesC programs are event-driven. Hardware interrupts trigger events whose effect will propagate through the interfaces. Event handling functions may also send command calls back down interfaces as part of the overall response to the initiating event. If an event is not time-critical, overall responsiveness may be improved by postponing the response until the processor is idle. Deferred execution is achieved by posting a task which responds to the event. Both synchronous and asynchronous commands are supported in nesC.

6.3 UML-Based Framework

In this section, we will introduce our proposed UML-based framework for TinyOS-based BSN systems. It consists of a UML-profile for TinyOS[62] through which designer can create reusable UML models, and store them in a TinyOS model repository. The repository stored models of both predefined components supported in nesC library and user-designed components. Methodology of applying our framework will be discussed in section 6.3.2.

6.3.1 UML Profile

The proposed UML profile specifies the conventions to build UML models of TinyOS-based BSN applications by defining stereotypes and tagged-values to capture domain specific information. In this profile, an input model is a set of UML 2.0 structural

diagrams, activity diagrams and interface state charts. Structural diagrams are used to model both modules and their configurations, while the interface state chart together with activity diagrams are used to model event-triggered behaviors. The UML profile lifts both structural and behavioral features of the design to UML level. This will give designers an overall view, abstracting away low level details.

6.3.1.1 Modeling Modules

The basic components in nesC program 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 `<<UserDefined>>` is given to UML classes that model user-created modules. Similarly, the `<<SystemDefined>>` stereotype is used to identify predefined modules supported by the nesC library. For a module that contains other components, a composite class will be needed. Commands and tasks are modeled as operations, and they are differentiated using 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* defines a distinct point of interaction between a class and the environment. The UML port takes the name of the nesC interfaces it specifies. It can have the following two types of interfaces:

- **Required interfaces:** These characterize the requests that can be made from the port's class (via the port) to its environment (external objects). A required interface is denoted by a socket notation.

- **Provided interfaces:** These characterize the requests that can be made from the environment to the class via a port. A provided interface is denoted by a lollipop notation.

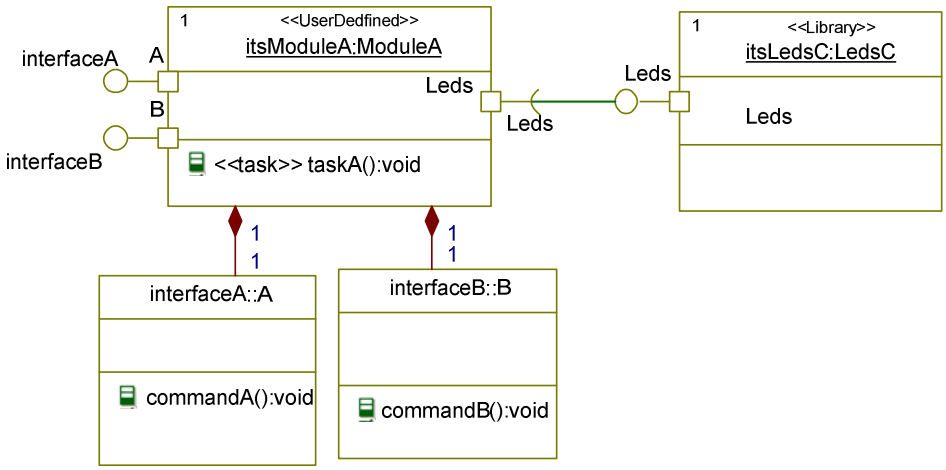


Figure 28: An example class diagram of tinyOS application[1]

By definition, the “required interfaces” model the "uses” interfaces of nesC modules, while, the “provided interfaces” model their “provides” interfaces. The “provide” and “required” properties identify the service directions, and the service definitions of nesC interfaces are captured in UML interface contracts.

As discussed in section 6.2, nesC commands are calls made on the service providers. Each provided service (command) has to be implemented in the owner

[1] The original drawings in the Rhapsody tool that we used can at times be too detailed and confusing. As such, some of the UML diagrams shown in the paper has been re-drawn and simplified for readability.

module. To do that, classes inherited from these interfaces are introduced to realize the detailed implementation. These implementation classes will share the same name with its implemented service (i.e., the command's name) and aggregated to the owner class. Figure 28 shows an example of the class diagram for a `moduleA`. Class A and B will implement the command A and command B of `moduleA` respectively. And A, B will provide contracts named `interfaceA` and `interfaceB`. Another example in the figure is that a system module `LedsC` provides a service interface `Leds`, while the implementation is hidden.

After a module is modeled, nesC code can be generated automatically from the model. The modules can be validated through simulation before they are added to a repository. This repository is a collection of UML models that can be reused. It forms the backbone of our framework and is the key to rapid prototyping and reuse.

6.3.1.2 Modeling New Application and Configuration

From the modules, a designer can start to build the application. Object diagram are predominantly used to capture structural information of objects. In nesC, inter-communication between modules is established by ‘wiring” the interfaces together and implementing the trigger actions. The application consists of two parts, an application module and a configuration for the application. UML object diagrams are well-suited to meet the modeling requirements here. A new module class with required interfaces is added, and the application configuration is represented by the whole object diagram. Predefined modules are added to the configuration, and they will provide the “required”

interfaces. The interfaces are connected using links, and the links correspond to the "wires" in nesC.

Figure 29 shows an object model diagram for the Blink example given in the nesC2.x library. It contains `BlinkC`, and three timers, each of which will trigger the LEDs with different frequencies. Among the components, `BlinkC` is the application module, and the rest are obtained directly from the repository. The "require" interfaces specified in `BlinkC` are connected to the "provided" interfaces of the timers, leds and the main routine.

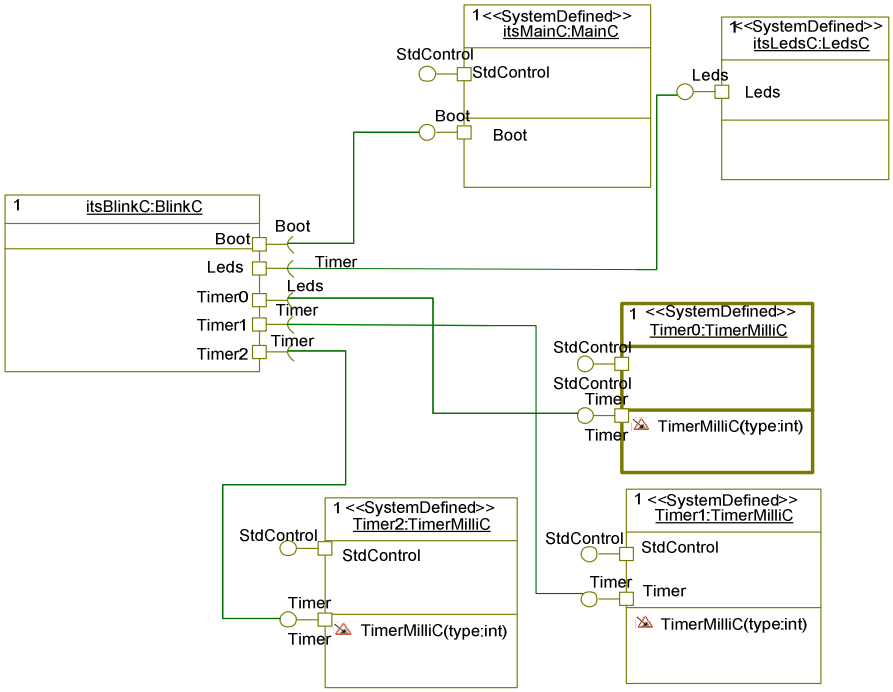


Figure 29: Object model diagram of Blink example

6.3.1.3 Modeling Behaviors

State charts are the natural choice for modeling objects' behavior.[72] In TinyOS, all the components are running concurrently and execution is preemptive. This is hard to model in a single state chart. TinyOS is a event-driven system, and all the events have to be sent through the corresponding interfaces. Moreover, nesC allows events with the same name. For example, if the system is connected to two separate sensors, both of them can have events with the same name, say `dataReady()` This is a potential source of confusion. On the other hand, if we use different state charts, one for each components, it will prevent the user from having an overall picture of the interaction between components, making the system's behavior hard to understand. Furthermore, this does not solve the problem of event ambiguity.

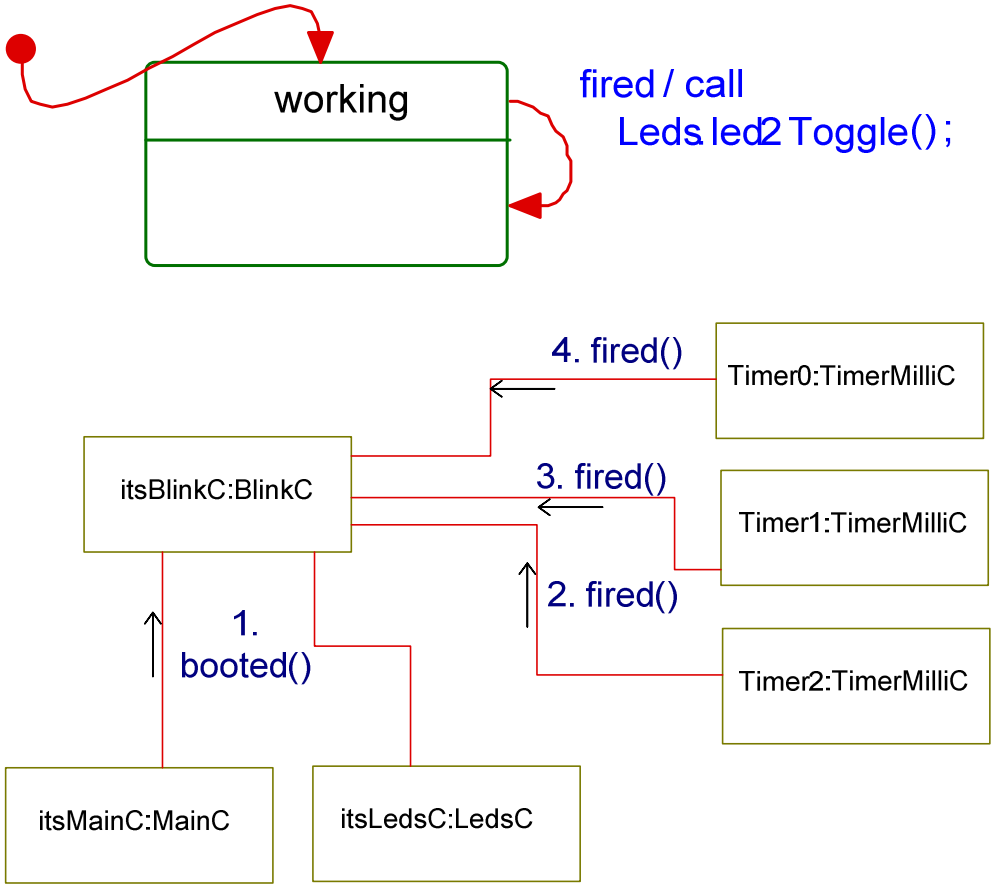


Figure 30: Interface state chart for Timer2.fired() and Collaboration diagram of Blink example

```
module BlinkC {
uses interface  Boot;
uses interface Timer<TMilli> as Timer0;
uses interface Timer<TMilli> as Timer1;
uses interface Timer<TMilli> as Timer2;
uses interface  Leds;
}
implementation {
event void Boot.booted(){
call Timer0.startPeriodic( 250 );
call Timer1.startPeriodic( 500 );
call Timer2.startPeriodic( 1000 );
}
```

```
event void Timer2.fired(){
call Leds.led2Toggle();
}
…
}}
```

Code 3: nesC code of Blink module

```
configuration BlinkAppC {}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;
  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

Code 4: nesC code of Blink configuration

To solve these problems, we use a combination of *collaboration diagrams* and local interface state charts to model system behaviors. Each structure diagram shall have a corresponding collaboration diagram to capture the interactions between the components. Using collaboration diagrams, the user will have a clearer picture of the interactions in the system. To the collaboration diagram, the designer can add the trigger-actions to the corresponding interface state chart. Each action specified on the association indicates a trigger event. The trigger actions will be specified using an interface state machine. Using the state chart attached to the interfaces will solve the ambiguity problem. Figure 30 shows the behavioral diagrams for Blink. In the collaboration diagram, 4 events are captured, namely, 3 *fired*, and 1 *booted*. The source

party of the actions is the initiator of the event, and the destination party of the actions shall take responses against these events. The state chart in Figure 30 captures the reaction of blinksC when it been triggered by event *fired()* from *Timer2*.

6.3.1.4 Code generator and implementation

Both the basic module and 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 to 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 interface state chart. Specifically, these are the actions embedded under the trigger action.

Code 3 shows a fragment of Blink module generated by our code generator. The used interfaces are generated from the links of structural model, and each event specified in collaboration diagram maps to an event trigger action, while the detailed implementation is generated from state diagram. For example, the event action of `Timer2` is derived from of interface state chart of UML port *Timer2*. Code 4 shows the configuration file of Blink. The connections of the interfaces are mapped into "wires" in nesC code. For each links in the structural diagram, a corresponding interface connection is made.

6.4 **Design Methodology**

The TinyOS module repository serves as a key role in the design process. It contains a set of UML classes formalized using our proposed UML profile. To design a new application, designer can either choose a similar application from repository or design from scratch. nesC library provided essential components such as timing, storage, ADC, and etc. All of system modules will be modeled as UML classes with stereotype <<*SystemDefined*>> and maintained in the repository together with user-define modules. The predefined module can facilitate user with a fast start. Figure 31 shows design flow using our proposed TinyOS repository. Designer will start to build new application by looking into the TinyOS module repository. If a new module is required, designer will need to build a new module by constructing UML class model using the UML profile. The newly defined module will be added to the repository, and the search continues until all the required modules are ready in the repository. Application will be build using structural diagram, simple select and drag can add the components into the application model. Links will be added to link the ports together which corresponding to the "wiring" in nesC. Because the TinyOS are highly event-oriented system, the behaviors of interfaces will be modeling in the corresponding interface state machine associated with the UML ports. Both module and application model will be used to generate nesC code. The generated code can be used for simulation or mote deployment.

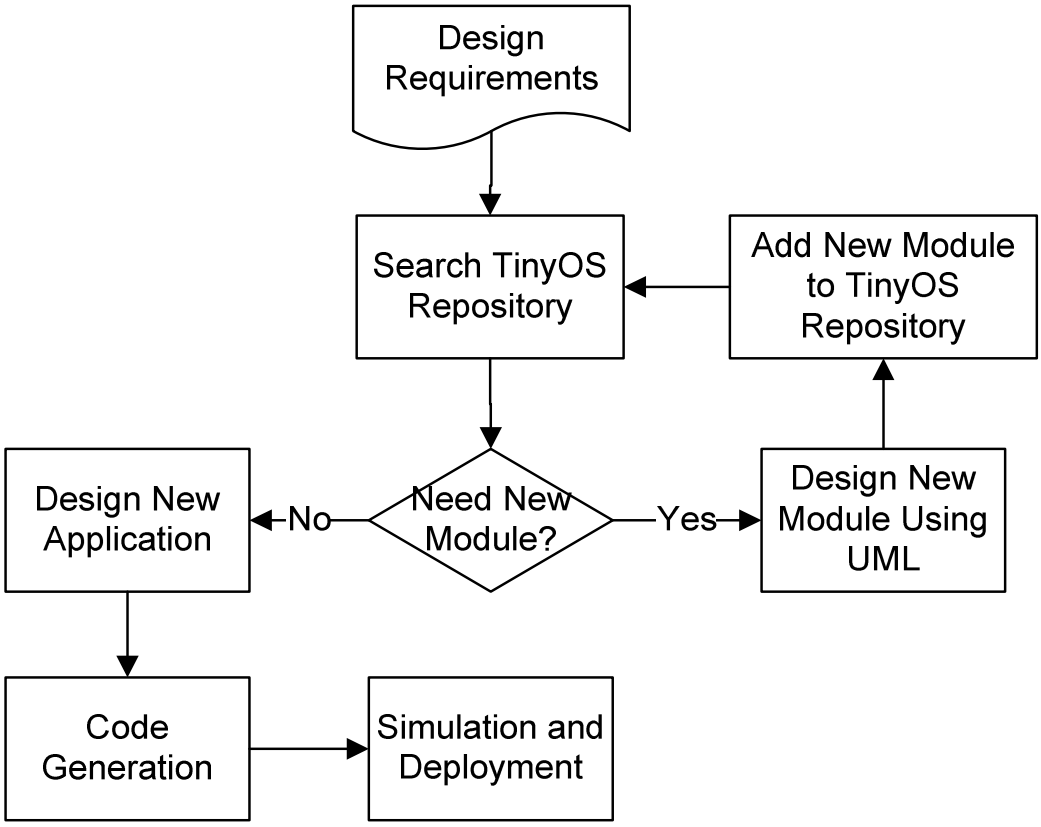


Figure 31: Design a new TinyOS application using our proposed framework

6.5 Case Studies

The ultimate goal of our framework is to create a user-friendly development environment for designer to create, modify and maintain their TinyOS applications. In this section, we will show how our framework can be applied in two cases studies.

6.5.1 Wheeze detection

Our first example is a microphone array used in a wheeze detection system. The system is a sound based health condition monitoring system and it is designed to detect, identify and selectively record the respiratory sounds of interest (e.g. wheezes). The

system consists of three parts: a sensor array and preprocessor, a mote, and a PDA. The application is designed to be running on TinyOS and periodically receiving the processed data from the data collection module. The data features are extracted and compared with the thresholds. If a clip of sound is classified as wheeze sound, it will be sent to PDA for storing.

To begin the design, we understand from specification that 1 microphone is used as sensor and it will be connected to ADC for data acquisition. The wheeze detection process will be controlled by a timer with period of 32ms. The detection results will be indicated by lighting on LEDs and sending out through wireless channel.

We will start the design by adding in predefined components: 1 timers(*MilliTimerC*), 1 ADC(*MSP430ADC12C*), 1 main routine(*MainC*), and 1 leds controller(*LedsC*). Object instances are created from each of the Modules. A control module, *WheezeDetectionC*, is then added in and marked using the stereotype `<<UserDefined>>`, and it will model the central controller who did all the sound classification. Properties such as buffers, and variables are written in as class properties, and other utility functions such detection, filtering, are added in as class methods. The implementation is in native C code. UML ports are added to WheezeDetectionC module to the services provided by those predefined modules, and they are connected to corresponding UML ports on the predefined modules.

Collaboration diagram is used to capture the interaction between the modules. We need to add in all the object instances into collaboration diagram. Lines and message calls are added if two modules have interactions. For example, event Boot

from MainC will be trigger the initialization of *WheezeDetection*, and the initialization code will be writing in the message call named Boot() from Main to Detector.

Our code generator takes in both structure and collaboration diagrams and produces nesC implementation. To estimate the timing and energy consumption, we execute the compiled nesC code on our simulator. When hardware is not ready to test, this could help us to optimize our application. We will show our tuning process in the experiment result section.

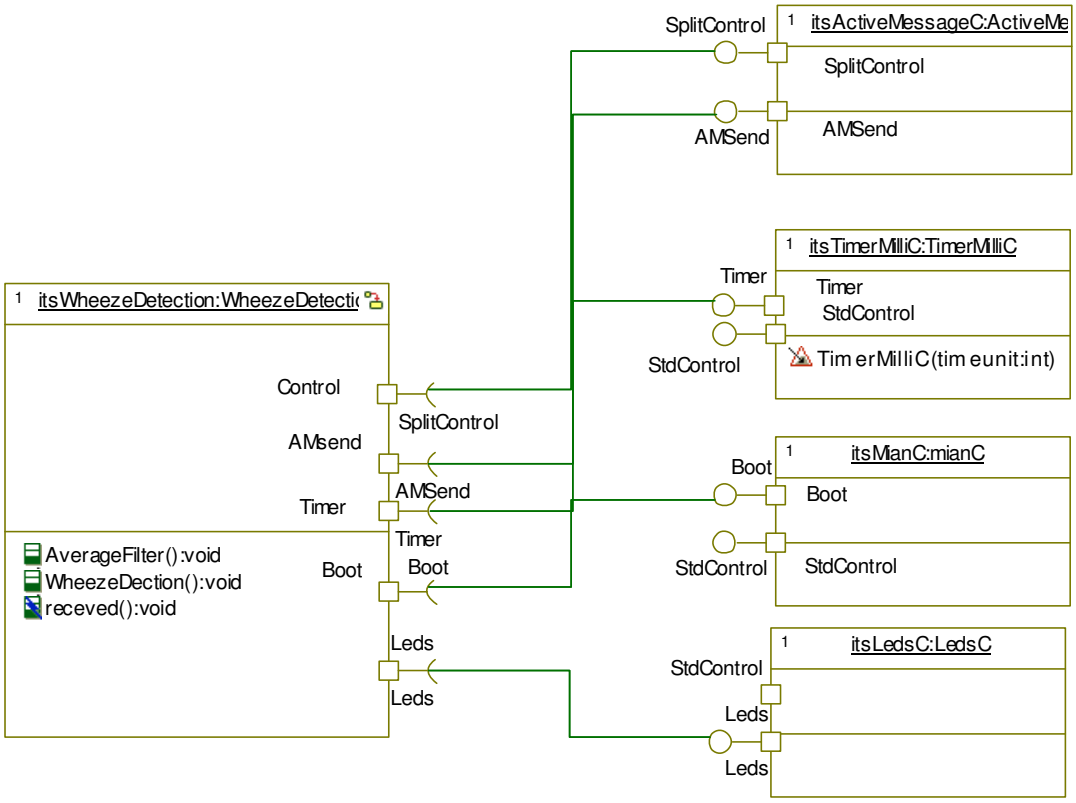


Figure 32: Object diagram of wheeze detection module

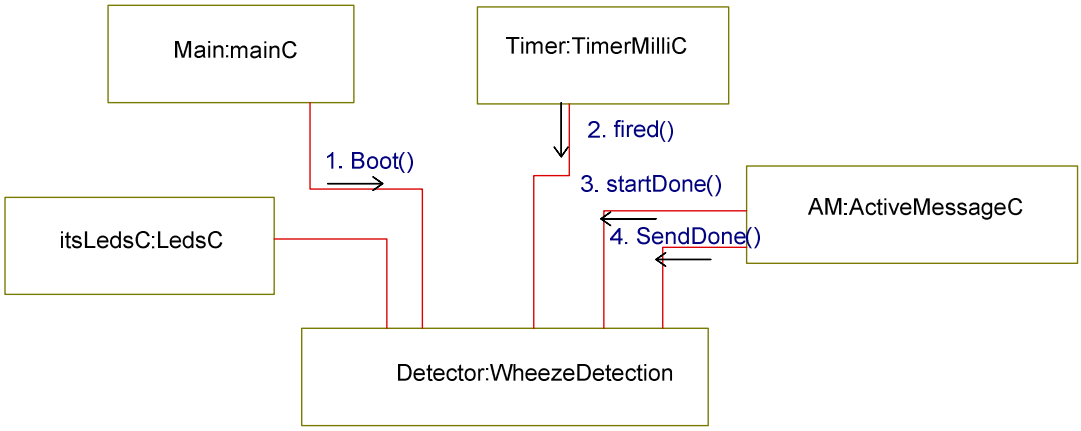


Figure 33: Collaboration diagram of wheeze detection application

6.5.2 ECG and SPO2 monitor

Our second example is a MEMSWear Biomonitoring application, ie. SpO2nECG application, taken from [19]. In this application, a single sensor mote collects data such as RED, IR and ECG from attached sensor at a sampling rate of 250Hz and then sends them all to the gateway station (PDA) for processing. The gateway station uses these data as input to compute heart rate, SpO2 and blood pressure. These outputs will be sent to the clinical department for further analysis. In case of emergency, a alarm will be raised to doctor and further action will be performed.

As the ECG and SPO2 sensors and their Analog to Digital Converter(ADC) are not available in the repository, we have to model them first. Figure 35 shows the structural diagram of ECGnSPO2 sensor, named as *PpgSensorC*. It has provided an interface named "*PpgSensor*". After construct the *PpgSensor* module, we have all the necessary modules in the repository. With similar approach described in section 6.2, we

use UML profile to construct the ECGnSPO2 application. We then build a new application by drawing a new structural diagram to capture the application. Figure 36 shows how the components are linked together. A class called *ECGnSPO2* will use necessary interfaces provided by system modules *Timer*, *ActiveMessage*, *Leds*, *Main* and user-defined module *PpgSensor*.

6.6 Experiment Results

During the experiments, we draw UML using Rhapsody, and a java program called UML2nesC is build to generate the nesC code. The generated code is then deployed onto BSN2.0 node from Empire College platforms[69] and BSN simulator for simulation. The generated applications produce identical result as compare to hand written code. Our experience also shows that building new application out of predefined modules using visualization tools can improve design efficiency and save effort of write code. Building applications such as Blink only takes a few minutes, while it may take dozens of minutes in typical way. Table 4 shows the application size and execution time of the code generator. From the table, the generator can generate the code within seconds. Table 5 shows the estimated implementation time of writing in hand and using our proposed framework. We can achieve 3-5times of speedup. In fact, during our design process, previous UML models can be reused. When we design ECGnSPO2 application, part of the structure model can be taken from Wheeze Detection application directly.

With aid of simulator, we are able to estimate the timing and energy consumption without the real hardware. WheezeDetection application has a tight time budget of 32ms to finish a detection cycle. The existing hardware platforms are not able to meet the timing requirement, and a new sensor node is under development with doubled processor speed. We adjusted the clock rate in simulator, and we can still keep our tuning work going before the hardware is made. Energy consumption is more of concern for ECGnSPO2 application, by executing ECGnSPO2 implementation, we are able to retrieve the detailed energy profile. We found that the wireless transmission takes more than 90% of the total energy consumption. Therefore, suggestions are made to put wireless transmitter into sleep mode while there are less workload. The top of Figure 34 expresses the original implementation consuming 39.58 mW (total energy/simulation time). With the proper adjustments, we could reduce 21% energy consumption comparing to the original design.

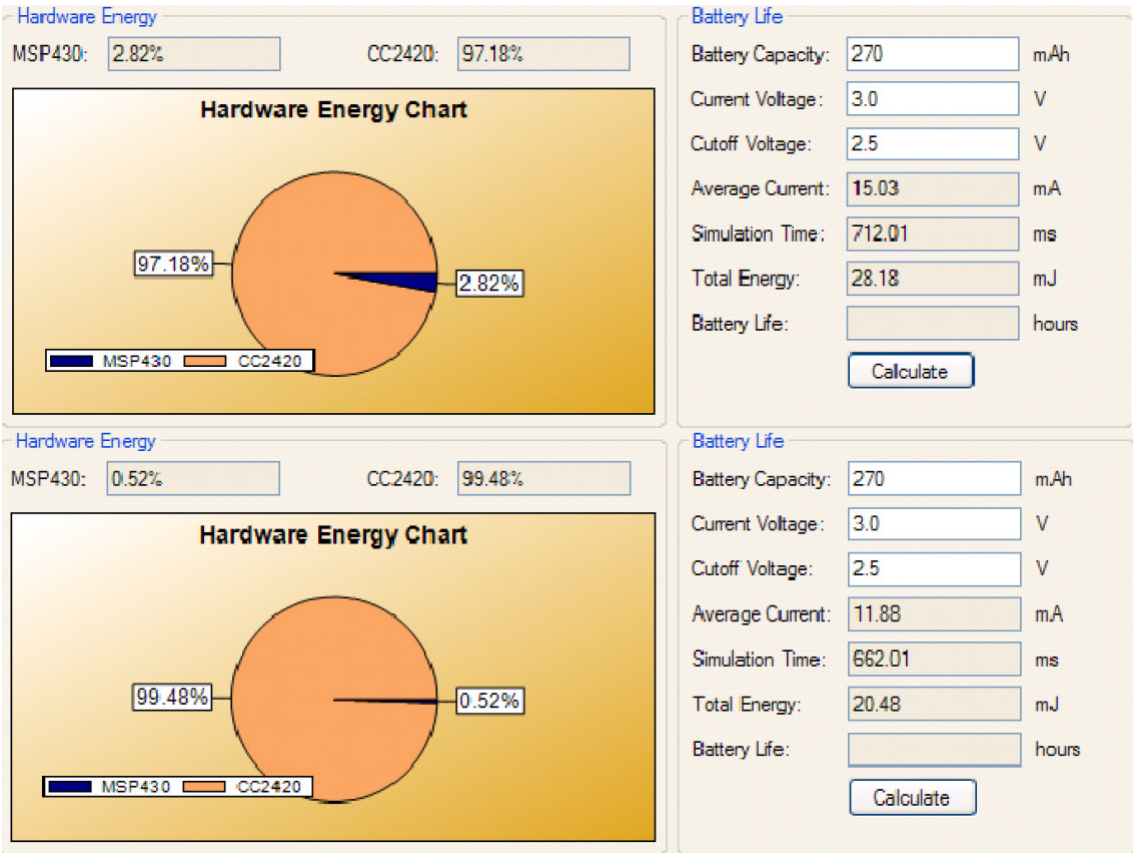


Figure 34: Experiment result of power consumption of the ECGnSpO2 application (a) hardware energy information of original design(top) (b) result after tuning (bottom)

Application	**Lines of generated nesC code**	**Time for code generation**
BlinkC	55	0.37 second
WheezeDetection	361	0.59 second
ECGnSPO2	1046	1.1 second

Table 4: Code Size and execution time of code generator

Application	**Written by hand**	**Design using UML**
BlinkC	30mins	<10mins
WheezeDetection	1 week	1working day
ECGnSPO2	2 weeks	2-3 working days

Table 5: Estimated implementation time

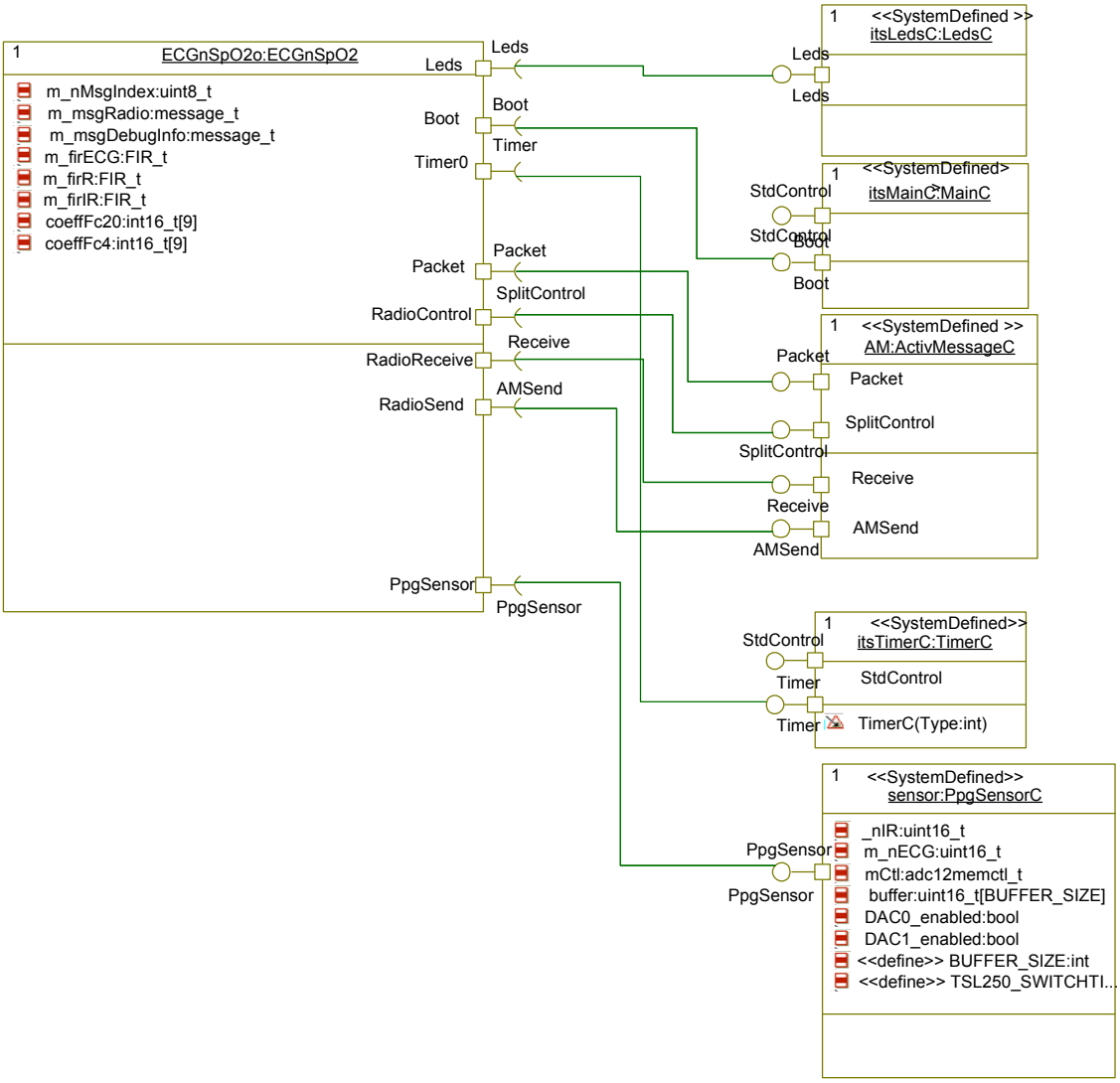


Figure 35: Structure diagram for ECGnSPO2 application

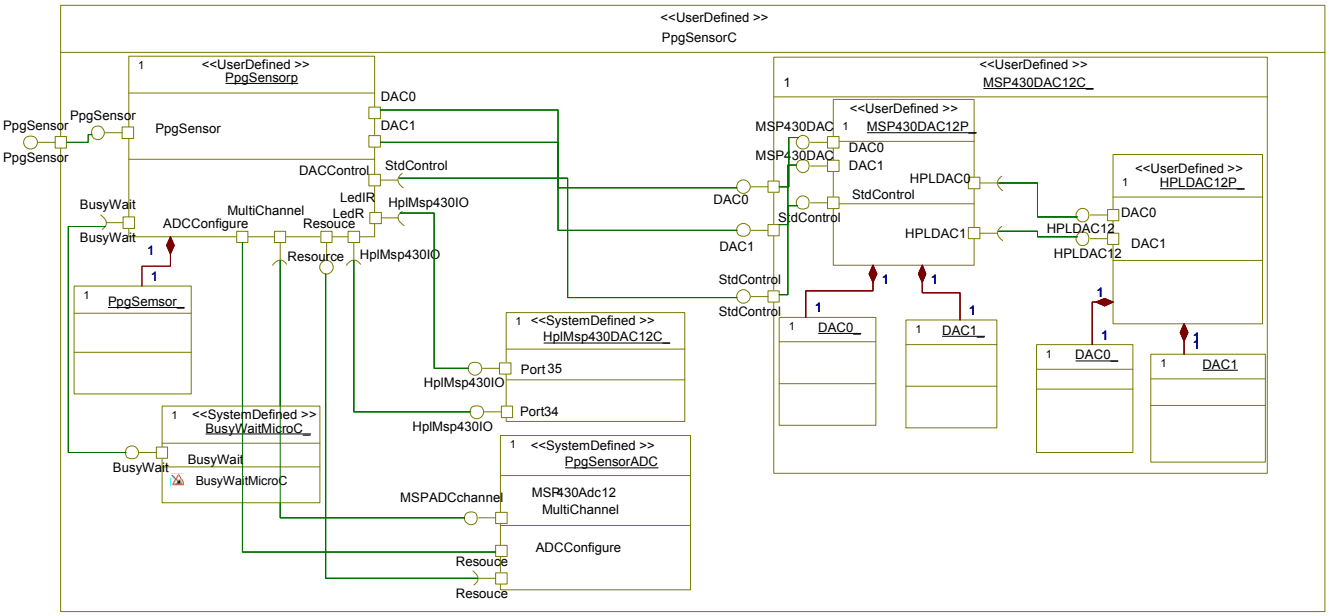


Figure 36: Class diagram for PpgSensor module

6.7 **Summary**

This chapter glues the previous chapters together and built a UML2.0-based framework to manage the complexity and reusability of TinyOS applications during pre-integration stage. The proposed framework consists of a UML profile for TinyOS and a corresponding simulation environment. The proposed UML profile abstract away the low-level details, and provide graphical high level models for designer to create and maintain their application. A TinyOS module repository, which contains UML models of the system and user defined module, will not only facility designer with quick development, but also encourages model exchanges. Customization of simulation environment is done at UML-level, and automatically customized simulator can be used to achieve accurate estimation of time and energy consumption. This will in turn facilitate the design space exploration and running. Our design flow will cut down design cost and effort greatly. Case studies over Body Sensor Network applications show that the complexity and reusability of application is well handled in the framework. And code generator has been proved to produce nesC code in a fast speed.

We summarize our design experiences in BSN application designs, and try to find for a design patterns. Another of the future works involves applying our framework to more complex platform such as WSN.

Chapter 7 Conclusion

In this thesis, we have presented a UML-based framework to accelerate the development of Body Sensor Network system design. Under this framework, UML plays a key role. Firstly, all the designs are done at UML level and it formalize the design specifications and facilitate the model exchange and reuse. Secondly, UML unified the design environment of BSN hardware and software components. Designers just need to focus on single. The approach not only save the design effort, but also enhance the reusability.

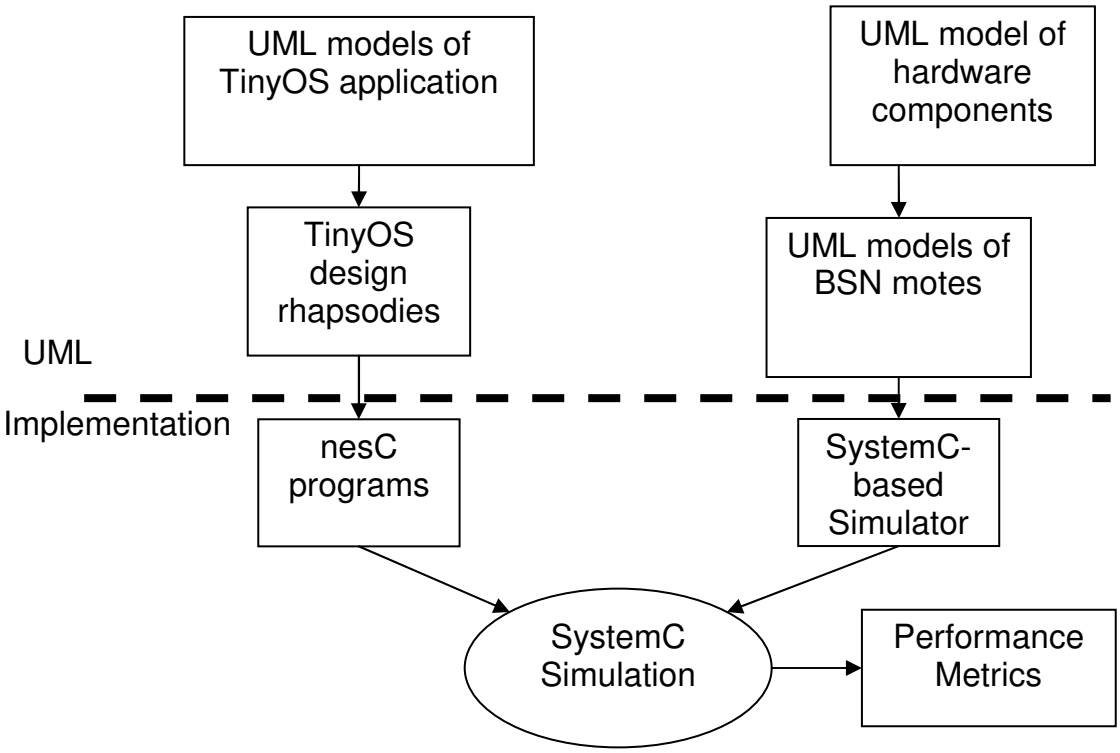


Figure 37: Summary of UML-based BSN design framework

Figure 37 summarized our framework. First, we have proposed a modeling language initial specification of requirement, structure and behavioral of a system. Both hardware and software components of the BSN application can be specified and designed with the profiles. UML has wide range of diagrams and notations for modeling different aspect of a system. We have chosen a subset of UML together with some extension mechanism. Among them, structure diagram is used to capture the structure of the system. Statecharts are used to model the components behaviors. We also use an activity diagram to capture the interactions between components interfaces.

With the chosen subset of UML, we have proposed unified modeling profiles for SystemC, SystemC-AMS. These profiles leverage the abstraction level of the SystemC design modules, and designers can focus on overall structure and behaviors of components rather than dive into the code level. Models are then refined to lower level specification. As we have shown, these UML profiles can capture the lower details of hardware system in easy maintain and exchange form. Using the automatic code generator, designer can get the implementation of their design as soon as they finish the refinement at UML level.

Using the profiles, a hardware Simulator to simulate the hardware process of BSN node are built. The simulator are the key component our framework. It serves as the linkage between software portion and hardware portion. Hardware designers will be able to customize BSN mode with the different sensors or processors. Predefined components (IPs) can be easily added or replaced in the simulator platform using the wrapped interfaces. Simulator can also execute the code which is under development,

and through the simulation can help the software designer to locate the bugs in the early stage. Moreover, the simulator servers as a pre-integration evaluation tools to the software designs. Design metrics such as timing and energy consumption, which can only be verified after the available of real hardware, can now be evaluated before the real integration.

We have also proposed a UML profile for TinyOS, the software operation system which is used by BSN nodes. This profile will lift the abstraction level of nesC implementation. The BSN designs often encounter similarity across deferent application. Patterns can be found for the time-triggered data collection and processing. In such situation, the basically components of the BSN application are often reusable. In our framework, these features are enhanced. We defined a model repository to collect existing BSN models, so that new designs can start by reusing/modifying existing models.

We have automated the model transformation process whereby all the motioned high level models can be converted to executable representation for simulation-based validation. This automation maintains the close bond between UML model to the implementation, and it greatly saves the implementation effort, and at the same time reduces the error. Automated code generation has been applied to both hardware and software portion of BSN designs. The purpose of automate model synthesis is to enable fast design realization and execution so that the performance factors can be verified at as early stage as possible.

Our practices of applying the framework to BSN applications show encouraging results. We are able to specify several BSN applications. With a few customizations, models can be reused cross these applications.

Our key results can be summarized as follows

- We have proposed UML profiles for both software and hardware components of BSN applications. With the profiles, we are able to do software and hardware design at much higher abstraction level. Automate code generation enforce the linkage between the high level models and lower level implementation, and this has relief the design from error-pruning coding, which lead to increase productivity and lower cost.
- A UML-modeled BSN simulator has been built with re-configurable components. Customization can be done at UML only, and the modified implementation can be automatically generated with our code generator. Interface synthesis ensures the "plug and play" style of interchangeability of new components.
- With help the customized simulator, we perform tests and performance evaluation before the real hardware commits. This pre-integration simulation is essential for the BSN software designers to tune their application code to meet the tight energy-computation requirements. The early stage refinement helps to reduce the re-work and debug effort in real software/hardware integration.

- We have made UML the single tool required in the co-design process of the BSN application.

7.1 Future works

In this thesis, we have outlined a framework to address co-design needs for Body sensor network designs. This framework unified the design tools of software and hardware component under the name of UML. A UML-modeled SystemC-based simulator enables rapid customization with “plug-in-and-play” tricks.

One direction for future work is adding in design space exploration tools. As we have presented in the chapter 5, performance metrics can be obtained from simulation results. This result can be used as inputs to design exploration tool, and thereby guild software/hardware refinement or system tuning to archive optimal solutions. Since the designs are abstract models and the code generation is automated, the exploration process can be semi-automated or even automated.

Another extension of our solution would be extending Body Sensor Network design principles to Wireless Sensor Networks (WSN). BSN normally has a centralized control station, while WSN often has a discrete infrastructure. By adding in networking specification, we should be able extend our framework to Wireless Sensor Network designs. We believe WSN designers can use our framework to address their co-design issues.

Bibliography

[1] A. Fehnker, L.F.W. van Hoesel, and A. H. Mader. Modelling and verification of the lmac protocol for wireless sensor networks., 2007.

[2] A. Sinha and A.P. Chandrakasan. JouleTrack: a web based tool for software energy profiling. In Proc. of the 38th Conference on Design automation (DAC), pp 220-225, New York, NY, USA. ACM. 2001

[3] A. Varga, The OMNeT++ discrete event simulation system, in Proceedings of the European Simulation Multi conference (ESM'2001), Prague, Czech Republic, June 6-9 2001, http://www.omnetpp.org/

[4] A. Varga. OMNet++: Discrete event simulation system.http://www.omnetpp.org/.

[5] Aurora, The AVR simulastion and analysis framework. http://compilers.cs.ucla.edu/avrora/, 2007

[6] B. Lin. Vercauteren S. Synthesis of concurrent system interface modules with automatic protocol conversion generation,ICCAD. pp. 101-108. 1994.

[7] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In Proc. of the 4th International Symposium on information Processing in Sensor Networks (IPSN), pp 67, Piscataway, NJ, USA. IEEE Press. 2005

[8] BSN node specification: http://vip.doc.ic.ac.uk/bsn/index.php?article=167

[9] C. Otto J. P. Gober and R. W. McMurtrey. A. Milenkovic, and E. Jovanov An implementation of hierarchical signal processing on wireless sensor in tinyos environment, 43rd Annual Southeast Regional Conference.2005.

[10] C.T Carr. Integration of UML and VHDL-AMS for Analogue system modeling. Formal Aspects of Computing. pp. 80-94. 2004

[11] D. Dietterle, J. Ryman, K. Dombrowski, R. Kraemer. Mapping of High-Level SDL Models to Efficient Implementations for TinyOS, Euromicro Symposium on Digital System Design (DSD'04). pp. 402-406. 2004

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. ACM SIGPLAN Notices.Vol. 38.pp. 1-11. 2003

[13] D. Panigrahi, S. Dey, R. Rao, K. Lahiri, C. Chiasserini, and A. Raghunathan Battery life estimation of mobile embedded systems, 14th International Conference on VLSI Design. 2001.

[14] D. Rakhmatov, S. Vrudhula, and D.A. Wallach A model for battery lifetime analysis for organizing applications on a pocket computer, IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2003.

[15] E. Cheong, E.A. Lee, and Y. Zhao, Viptos: a graphical development and simulation environment for TinyOS-based wireless sensor networks. In Proceedings of the 3rd international Conference on Embedded Networked Sensor Systems, San Diego, California, USA, November02 - 04, pp.302-302. 2005

[16] E. Christen Vakalar K. VHDL-AMS a hardware description language for analog and mixed-signal applications, IEEE trans. On circuit and systems.1999.Vol. 46.pp. 1263-1272. 1999

[17] E. Feig, and Linzer, E. Discrete cosine transform algorithms for image data compression, Proceedings of Electronic Imaging '90 East .pp. 84-87. 1990

[18] E. Markert M. Dienel, G. Herrmann, U. Heinkel, SystemC-AMS Assisted Design of an Inertial Navigation System, Sensors Journal, IEEE 2007.Vol. 7. 2007

[19] F. E. H. Tay D. G. Guo, L. Xu, M. N. Nyan, and Yap K. L Memswear-Biomonitoring System for Remote Vital Signs Monitoring, 4th International Symposium on Mechatronics and its Applications (ISMA07). 2007.

[20] F. O¨sterlind. The COOJA simulator - user manual. http://www.sics.se/ fros/cooja.

[21] G. Borriello, P. Chou and R. Ortega. Embedded System Co-Design: Towards Portability and Rapid Integration, Hardware/Software Co-design, NATO ASI Series. pp. 1-28. 1996

[22] G. Sachdeva R. D¨omer, and P. Chou System modeling a case study on a wireless sensor network, : Technical Report CECS-TR-05-12 / University of California.2005.

[23] G. Y. Jeong K. H. Yu, and Kim. N. G Continuous blood pressure monitoring using pulse wave transit time, International Conference on Control, Automation and Systems (ICCAS).2005.

[24] G. Casinovi and C. Young. Estimation of power dissipation in switched-capacitor circuits, IEEE Trans. on CAD of Integrated Circuits and Systems.pp. 1625–1636. 2003

[25] G.Z. Yang. Body Sensor Networks. Springer-Verlag New York, Inc. 2006.

[26] GRATIS: http://w3.isis.vanderbilt.edu/Projects/nest/gratis/GratisIITechOver.html

[27] H. Wada P. Boonma and J. Suzuki. Modeling and Executing Adaptive Sensor Network Applications with the Matilda UML Virtual Machine. 11th IASTED International Conference on Software Engineering and Applications (SEA).Cambridge, MA, 2007.

[28] H.Y. Tyan. Design, realization and evaluation of a component-based compositional software architecture for network simulation, Ph.D. dissertation, 2002. see also http://www.j-sim.org

[29] J.B. Schmitt, F.A. Zdarsky, and L. Thiele. A comprehen-sive worst-case calculus for wireless sensor networks with in-network processing. In RTSS 07: Proceedings of the 28th IEEE International Real-Time Systems Symposium, pages 193¨C202, Washington, DC, USA. IEEE Computer Society. 2007

[30] J.B. Schmitt and U. Roedig. Sensor network calculus - a framework for worst case analysis. Distributed Computing in Sensor Systems, 3560:141¨C154, 2005.

[31] J. Eriksson, A. Dunkels, N. Finne, F. Osterlind, and T. Voigt. MSPsim - an extensible simulator for msp430-equipped sensor boards. In Proc. ofthe European Conference on Wireless Sensor Networks (EWSN), 2007.

[32] J. Zhu J. MetaRTL: Raising the abstraction level of RTL design, Design Automation and Test in Europe.Munich, Germany, 2001.

[33] K. Hung, Y. T. Zhang, and B. Tai Wearable medical devices for telehome healthcare, 26th Annual International Conference on the IEEE EMBS.2004.

[34] K. Virk, K. Hansen, and J. Madsen. System-level modeling of wireless integrated sensor networks, 2005 International Symposium on System-on-Chip. 2005.

[35] K.D. Kathy etc. Fast and Accurate Simulation of Biomonitoring Applications on a Wireless Body Area Network. 5th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2008).2008.

[36] K.D. Nguyen, I. Cutcutache, S. Sinnadurai, S. Liu, B. Cihat, A. Curic, T.B. Tok, X. Lin , F.E.H. Tay, and T. Mitra. A systemc-based fast simulator for biomonitoring applications on wireless ban. In Workshop on Software and Systems for Medical Devices and Services 2007 (SMDS 2007),December 2007.

[37] K.D Nguyen, Z. Sun, P.S. Thiagarajan, and W.F. Wong, Model-driven SoC Design Via Executable UML to SystemC, Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS).pp. 459-468. 2004

[38] K.D. Nguyen Z. Sun, P.S. Thiagarajan, and W.F. Wong UML for SOC Design , edited by Müller Grant Martin and Wolfgang.pp. 175-197. 2005.

[39] K. Fall and K. Varadhan. The network simulator NS-2. http://www.isi.edu/nsnam/ns/.

[40] K. Lüth, T. Peikenkamp, and J. Niehaus. HW/SW Cosynthesis using Statecharts and Symbolic Timing Diagrams. In Proceedings of the 9th IEEE International Workshop on Rapid System Prototyping, Juni 1998.

[41] K.S. Chung. Gupta, R.K., Liu, C.L. An algorithm for synthesis of system-level interface circuits, Computer-Aided Design, IEEE/ACM International Conference.pp. 442447. 1996

[42] L. Breslau D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu Advances in network simulation, IEEE HLDVT.2000

[43] L. Carloni Passerone, R., Pinto. Languages and Tools for Hybrid Systems Design, Foundations and Trends in Design Automation.pp1-204. 2006

[44] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks, in USENIX 2004 Annual Technical Conference, pp. 283–296. 2004

[45] L. Lavagno G. Martin, and B. Selic. UML for Real: Design Embedded Real-Time Systems: Kluwer Academic Publishers, 2003.

[46] L. Lopes, F. Martins, M.S. Silva, and J. Barros. A formal model for programming wireless sensor networks. In Proc. of the International Conference on Distributed Computing in Sensor Systems (DCOSS), 2007.

[47] L.M. Reyneri. An Object Oriented Codesign Flow for low-cost HW/SW/mixed-signal systems based on UML, Electrotechnical Conference 2006. pp:80-84. 2006.

[48] L. Younes K. Meriam, D. Ayoub. System on Chips optimization using ABV and automatic generation of SystemC codes ,Microprocessors & Microsystems, November 2007.7 : Vol. 31. 2007

[49] Labview software. http://www.ni.com/labview/, 2007.

[50] M. Alassir, J. Denoulet, O. Romain, and P. Garda. A SystemC AMS model of an 12C bus controller. Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference, Sept. 5-7, 2006, pp: 154-158, 2006

[51] M. Fruth. Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol. In Proc. of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), 2006

[52] M. Karir. Atemu - sensor network emulator / simulator / debugger. http://www.hynet.umd.edu/research/atemu/.

[53] M. Keating, and P. Bricaud, Reuse methodology manual from system-on-chip designs ,Kluwer Academic publishers, 2002

[54] M. Vasilevski Pecheux, F. Aboushady, H. de Lamarre, L. Modeling heterogeneous systems using SystemC-AMS case study: A Wireless Sensor Network Node, Behavioral Modeling and Simulation Workshop 2007(BMAS 2007), IEEE International. pp. 11-16. 2007

[55] Matlab, http://www.mathworks.com/.

[56] Moving Picture Experts Group, http://www.mpeg.org.

[57] N. Fournel A. Fraboulet, G. Chelius, E. Fleury, B. Allard, and O. Brevet Worldsens: from lab to sensor network application development and deployment, 6th International Conference on Information Processing in Sensor Networks.pp. 551–552. 2007

[58] O. Landsiedel, K. Wehrle, B. Titzer, and J. Palsberg. Enabling detailed modeling and analysis of sensor networks. Praxis der Informationsver-arbeitung und Kommunikation, pp.101-106, April 2005.

[59] OCP-IP, http://www.ocp-ip.org.

[60] P. Baldwin S. Kohli, E. A. Lee, X. Liu, and Y. Zhao Modeling of Sensor Nets in Ptolemy II., Int'l Sym. on Information Processing in Sensor Networks.2004.

[61] P. Chou, G. Ortega, G. Borriello, Interface co-synthesis techniques for embedded systems, ICCAD. San Jose, 1995.

[62] P.Levis TinyOS: An operating system for sensor networks, Ambient Intelligence W. Weber, J. Rabaey, and E. Aarts, Eds., Berlin:Springer. pp. 115–148. 2005

[63] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyos applications. In Proc. of the 1stACM Conference on Embedded Networked Sensor Systems (SenSys). 2003.

[64] P. Nikitin V. Normark E., Wakayama C., and Shi C.-J. R. VHDL-AMS modeling and simulation of a BPSK transceiver system, Proc. of IEEE International Conf. on Circuits and Systems for Communications. 2004.

[65] R. Chandra IP-Reuse and platform base designs, system level design with embedded platforms, DAC Tutorial. 2000.

[66] R. Damasevicius and V. Stuikys. Soft IP customization models based on high-level abstractions, Information Technology and Control.Kaunas, Technologija.Vol. 34.pp. 125-134. 2005

[67] R. Mukherjee, Jones, A., and Banerjee, P. System level synthesis of multiple IP blocks in the behavioral synthesis tool, Int. Conference on Parallel and Distributed Computing and Systems (PDCS).2003.

[68] Rhapsody, http://www-01.ibm.com/software/awdtools/rhapsody/.

[69] S. Escolar J. Carretero, et al. A driver model based on Linux for TinyOS, IEEE Symposium on Industrial Embedded Systems.pp. 361-364. 2007

[70] S. Klaus, S. A. Huss and T Trautmann, Automatic Generation of Scheduled SystemC Models of Embedded Systems From Extended Task Graphs, Proc. Int. Forum on Design Languages, Marseille ,France, September 2002

[71] S. Park, A. Savvides, and M. B. Srivastava. SensorSim: A simulation framework for sensor networks. In Proc. of the 3th ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM), 2000.

[72] S. Smolau, R. Beaubrun. State-oriented programming for TinyOS, Proceedings of the 2007 summer computer simulation conference.pp. 766-771. 2007

[73] S. Sundresh, W. Kim, and G. Agha. SENS: A sensor, environment and network simulator. In Proc. of the 37th Annual Symposium on Simulation (ANSS), page 221, Washington, DC, USA. IEEE Computer Society. 2004

[74] S. Tschirner, L. Xuedong, and W. Yi. Model-based validation of QoS properties of biomedical sensor networks. In Proc. of the 7th ACM international

conference on Embedded software (EMSOFT), pages 69¨C 78, New York, NY, USA, 2008.

[75] SystemC home. http://www.systemc.org.

[76] SystemC-AMS home. http://www.systemc-ams.org.

[77] T. Cutcutache, N. Dang, W.K. Leong, S. Liu, K.D. Nguyen, L.T.X. Phan, E.J. Sim, Z. Sun, T.B. Tok, L. Xu, F.E.H. Tay, and W.F. Wong BSN Simulator: Optimizing Application Using System Level Simulation, 6th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2009).pp. 9-14 .2009

[78] T. K. Tan I, A. Raghunathan, and N. K. Jha. Emsim: An energy simulation framework for an embedded operating system, 2002.

[79] T. Kun, Wang, H., and Bian, J.N. A generic interface modeling approach for SOC design, ICSICT'04.Beijing. pp. 1400-1403. 2004

[80] UML documentation, http://www.uml.org.

[81] UML profile specification, – http://www.omg.org/technology/documents/profile_catalog.htm.

[82] UML-SOC homepage, http://www.c-lab.de/uml-soc .

[83] V. Shnayder, M. Hempstead, B. Chen, G.W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications In Proc. of the 2st ACM Conference on Embedded Networked Sensor Systems (SenSys)., 2004.

[84] V. Sinha, F. Doucet, C. Siska, R. K. Gupta, S. Liao, A. Ghosh. YAML: A Tool for Hardware Design Visualization and Capture. In Proc. International Symposium on System Synthesis, 2000

[85] V. Stuikys, R. Damasevicius. Soft IP customization model based on metaprogramming techniques, Informatica, Lith. Acad. Sci..2004.pp. 111-126.

[86] Velocity website, http://velocity.apache.org/.

[87] W. Fornaciari, F. Salice, P. Micheli and L. Zampella. A First Step Towards Hw/Sw Partitioning of UML Specifications. IEEE/ACM Design Automation and Test in Europe (DATE'03), Munich, Germany, p.668-673. March, 2003.

[88] W.H. Tan, P.S. Thiagarajan,W.F. Wong, Zhu Y. and Pilakkat S.K. Synthesizable SystemC Code from UML Models, International Workshop on UML for SoC Design (USOC 2004). 2004.

[89] Wikipedia, Body sensor network, http://en.wikipedia.org/wiki/Body_sensor_network

[90] Wikipedia, Embedded system, http://en.wikipedia.org/wiki/Embedded_system.

[91] Y. Liang, A. Roychoudhury, and T. Mitra. Timing analysis of body area network application, 7th International Workshop on Worst-Case Execution Time Analysis (WCET).2007.

[92] Y. Liu, B. Veeravalli, and S. Viswanathan. Critical-path based lowenergyscheduling algorithms for body area network, 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).2007.

[93] Y. Zhu, Z. Sun, A. Maxiaguine, and W.F. Wong. Using UML 2.0 for System Level Design of Real Time SoC Platforms for Stream Processing, 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.Hong Kong, pp. 154-159. 2006

[94] Z. Guo, A. Mitra, and W. Najjar. Automation of IP core interface generation for reconfigurable computing, Int. Conference on Field Programmable Logic and Applications (FPL 2006).Madrid, Spain , 2006.

[95] Z. Sun and W.F. Wong. A UML-Based Approach for Heterogeneous IP Integration, 14th Asia and South Pacific Design Automation Conference (ASP-DAC).Yokohama , pp. 155-160. 2009.

[96] Z. Sun, C-T. Ye, and W.F. Wong A UML 2-based HW/SW Co-Design Framework for Body Sensor Network Applications, Design, Automation, and Test in Europe (DATE 11).Grenoble. pp. 1505-1508. 201

[97] Z. Sun, Y. Zhu, W.F. Wong, and S.K. Pilakkat, Design of Clocked Circuits using UML, Asia and South Pacific Design Automation Conference 2005 (ASP-DAC).ShangHai, 2005.