Designing Power Aware Wireless Sensor Networks Leveraging Software Modeling Techniques

John Khalil Jacoub

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

at

University of Ontario
Institute of Technology
March 2014

Abstract

Wireless Sensor Networks (WSNs) are typically used to monitor specific phenomena and gather the data to a gateway node, where the data is further processed. WSNs nodes have limited power resources, which require developing power efficient systems. Additionally, reaching the nodes after a deployment to correct any design flaws is very challenging due the distributed nature of the nodes. The current development of WSNs occurs at the coding layer, which prevent the design from going through a typical software design process. Designing and analyzing the software modules of a WSN system at a higher abstraction layer than at the coding level will enable the designer of a WSN to fix any design errors and improve the system for power consumption at an early design stage, before the actual deployment of the network.

This thesis presents multiple Unified Modeling Language (UML) design patterns that enable the designer to capture the structure and the behavior of the design of a WSN at higher abstraction layers. The UML models are developed based on these design patterns that are capable of early validation of the functional requirements and the power consumption of the system hardware resources by leveraging animation and instrumentation of the UML diagrams.

To support the analysis of power consumption of the communication components of a WSN node, the Avrora network simulator was integrated with the UML design environment such that designer is able to analyze the power consumption analysis of the communication process at the UML layer. The UML and the Avrora simulation integration is achieved through developing a code generator that produces the necessary configuration for Avrora simulator and through parsing the simulator results. The methodology presented in this thesis is evaluated by demonstrating the power analysis of a typical collector system.

Keywords: Wireless Sensor Networks, Power Consumption and Analysis, Design Patterns, UML System Modeling, UML model execution

Author's Statement

I, John Khalil Jacoub, hereby certify that I have prepared this Ph.D. thesis independently, and that only those sources, aids, and advisors that are duly noted herein have been used and/or consulted.

Signed this 25th day of April, 2014:

John Khalil Jacoub

Acknowledgments

I would like to thank my supervisor Dr. Ramiro Liscano for giving me the opportunity to pursue my Ph.D. degree under his supervision. I would like to express my sincere gratitude and appreciation for Dr. Liscano for his valuable advice, time, and encouragement he provided during these past five years. I would to extend my appreciation to my co-supervisor Dr. Jeremy Bradbury for his guidance and helpful recommendations throughout the thesis work. Thank you to the University of Ontario Institute of Technology (UOIT) and the Faculty of Engineering for offering me the facilities to complete my research.

A very special thank you to my lovely wife, Mrs. Laura Jacoub, for her love, support, patience, and faith in me. She has played an important role in revising and enhancing the thesis structure. Her support and encouragement were always motivating me to complete the thesis. Thank you for standing by me as I worked long hours to complete this degree.

I also wish to thank my parents, Dr. Khalil Jacoub and the late Mrs. Magda Jacoub, for their support, love, and for guiding me to be the man I am today. My parents have always been a source of encouragement and supported me to pursue higher education. I would like to extend my gratitude to my brother Dr. George Jacoub and my sister-in law Mrs. Michaela Jacoub for guiding me to apply for Ph.D. opportunities in Canada and for helping me to establish my life here.

Finally, but most importantly, I thank and praise my Lord Jesus Christ for His support spiritually and unwavering help every day in all conditions. Thank you, God, for blessing me and giving me the strength and perseverance to complete my Ph.D. degree. For "*I can do all things through Christ who strengthens me*" Philippians 4:13.

Content

Abstract ii

Author's Statement iv

Acknowledgments v

Content vi

List of Tables xiii

List of Abbreviations and Symbols xiv

1. Introduction 1

1.1 Motivation 1

1.2 Thesis Statement and Scope of Research 3

1.2.1 Problem Statement 3

1.2.2 Thesis Statement 3

1.3 Thesis Methodology and Contribution 3

1.3.1 Thesis Methodology 3

1.3.2 Thesis Contribution 6

1.4 Tools leveraged in the thesis 7

1.4.1 IBM Rational Rhapsody 7

1.4.2 Avrora Simulator 10

1.5 Thesis Structure 13

2. Background 16

2.1 WSN Structure 16

2.2 WSN Power Consumption 18

2.3 WSN Modeling 19

3. Modeling Techniques for Wireless Sensor Networks 21

3.1 Introduction 21

3.2 Modeling Techniques Overview 22

3.2.1 High-Level SDL Models (HL-SDL) 22

3.2.2 Insense 23

3.2.3 MathWorks Modeling Approach 23

3.2.4 Model Driven Engineering Approach (MDEA) 24

3.2.5 Promela and the SPIN Model Checker for WSNs (PM) 25

3.2.6 SensorML 25

3.2.7 SystemC-AMS 26

3.2.8 UM-RTCOM Model 26

3.2.9 eXtended Reactive Modules (XRM) 27

3.3 Overview from the Node's Software Architecture Perspective 27

3.4 Overview from the System's Architecture Perspective 29

3.5 Model and Simulation Integration 30

3.6 Concluding Remarks for the Review 31

4. UML Modeling and Power Consumption Analysis for Wireless Sensor Networks . 35

4.1 Introduction 35

4.2 WSN Systems Design Architecture 36

4.3 Design Patterns for Wireless Sensor Networks 39

4.3.1 Sensor Node Service Component Design Pattern 39

4.3.2 Sensor Node Event Handler Design Pattern 42

4.3.3 Association of the Application Software and OS Support Components 46

4.4 State-Chart Stereotypes 49

4.5 Analysis in the UML Model Layer 50

4.5.1 Instrumentation of State-Charts for Power and Timing Annotation 51

4.5.2 Model Execution 53

4.6 Summary and Limitations 57

5 Generating TinyOS code for UML High Level Models 60

5.1 Introduction 60

5.2 TinyOS Code Structure 61

5.2.1 Component Wiring 62

5.2.2 Event Handlers 63

5.3 Code Generator Structure 65

5.3.1 XML Parser 65

5.3.2 Database Structure 66

5.3.3 Code Builder 68

5.4 Instrumentation of the Generated Code 70

5.4.1 Code validation instrumentation 70

5.4.2 Instrumentation to Facilitate the Integration of the Simulator to the UML model……….......... 71

5.5 Code Generation and Instrumentation Example 73

5.6 Validating the generated Code 77

5.7 Summary 81

6 Integration of the Simulator and the UML Models 83

6.1 Introduction 83

6.2 The Parsing Tool 84

6.3 Integrating Simulation results with UML Model Example 87

6.4 Summary 89

7 Evaluation 90

7.1 Introduction 90

7.2 Power Consumption Improvement 91

7.3 SensIV System Design 93

7.3.1 SensIV System Requirements and Specifications 93

7.3.2 UML Models for SensIV System 94

7.3.3 SensIV Analysis at the Modeling Layer 100

7.3.4 Power Consumption Analysis of the WSN Network 102

7.3.5 Network Analysis Accuracy 122
7.4 Framework Automation 122
7.5 Framework Generality 124
7.5.1 UML Model and Code Generator Generality 125
7.5.2 The Generality of the Leveraged Tools 126
7.6 Threats to Validity 127
7.6.1 Monitored Data Type 127
7.6.2 Data Collection Protocols 128
7.6.3 Fault Tolerance Capability 128
7.6.4 Hardware Constraints 129
7.6.5 Network Topology 129
7.6.6 Software Design 130
7.7 Summary 131
8 Summary and Conclusion 133
8.1 Summary 133
8.2 Thesis Contribution 133
8.3 Future Work 135
9 Reference I
10 Appendices VIII

List of Figures

Figure 1-1: Thesis Methodology Diagram 13

Figure 4-1: Modeling, Verification, and Analysis at the Model Layer 36

Figure 4-2: State-chart of Sensor Hardware Module 42

Figure 4-3: Split Phase Communication 44

Figure 4-4: Portion State-Chart for SensIV System 46

Figure 4-5: RFID System Components 48

Figure 4-6: ADC State-Chart Model 52

Figure 4-7: Sensing State-Chart Model 52

Figure 4-8: Trigger Sensors State-Charts 53

Figure 4-9: Executed Sequence Diagram- Interaction between Application and Sensor Component 54

Figure 4-10: Sequence Diagram Comparison Results-1 56

Figure 4-11: Sequence Diagrams Comparison Result-2 57

Figure 5-1: Code Generation and Verification 61

Figure 5-2: TinyOS components 63

Figure 5-3: Code Generator Structure 64

Figure 5-4: Database Structure 66

Figure 5-5 (A): The Code Instrumentation Example 70

Figure 5-6 (B): The Model States 71

Figure 5-7: Example of Debugging Statement for Simulator Integration 73

Figure 5-8: The Trace File Output 73

Figure 5-9: Code Generation Example 76

Figure 5-10: Executed Sequence Diagram for Counter Sender Node (Part-A) 79

Figure 5-11: Executed Sequence Diagram for Counter Sender Node (Part-B) 80

Figure 5-12: The screen shot Node1 log file.. 81

Figure 6-2: Parsing Log files and Trace file 86

Figure 6-3: Power Consumption Analysis of Counter Application 88

Figure 7-1: Service Components UML Model for SensIV 95

Figure 7-2: Routing OS Component State-Chart.. 96

Figure 7-3: Portion of SensIV State-Chart.. 98

Figure 7-4: SensIV Panel Diagram 99

Figure 7-5: Power Analysis for Three Scenarios of Controlling the ADC 103

Figure 7-6: SensIV Topology Structure 105

Figure 7-7: Physical Location of SensIV Nodes... 105

Figure 7-8: The Simulator Configuration 106

Figure 7-9: The Call Trace Produced by Avrora 108

Figure 7-10: SensIV Executed Sequence Diagram...112

Figure 7-11: Design Modification for Event Structure 115

Figure 7-12:Aggregation SensIV State-Chart Design 118

Figure 7-13: Power Consumption of Network Sending Process 119

Figure 7-14: Power Consumption for the Network Receive Processing Process 120

Figure 7-15: Enabling LPL in Booted Event 121

Figure 7-16: Power Consumption of Receiving Energy for LPL Vs Normal Mode 121

List of Tables

Table 3-1: Overview of the Modeling Techniques 33

Table 3-2: Modeling of Node and System Arcitecture 33

Table 4-1: Model Stereotypes 50

Table 5-1: Explanation of the Database Tables 68

Table 7-1: Power Consumption Constrains 101

Table 7-2: Power Consumption Analysis for SensIV Determintsic Components 103

Table 7-3: SensIV Nodes Total Power Consumption 113

Table 7-4: Power Consumption Report for Node 1, 6, 8, and 9 113

Table 7-5: Power Consumption for Nodes 1, 6, 8, and 9 after Modification 115

Table 7-6: Aggregation Power Consumption for Nodes1, 6, 8, and 9 119

Table 7-7: Designer and Automated Intervention Comparison 124

List of Abbreviations and Symbols

WSN	Wireless Sensor Networks
ADC	Analog-to-Digital Converter
IP	Internet Protocol
UML	Unified Modeling Language
XML	Extensible Markup Language
LQI	Link Quality Indicator
RSSI	Received Signal Strength Indication
SDL	Specification and Description Language
TLC	Target Language Complier
PIM	Platform Independent Model
MDEA	Model Driven Engineering Approach
WSAN	Wireless Sensor Actor Networks
VM	Virtual Machine
XRM	eXtended Reactive Modules
RM	Reactive Modules
WECT	Worst Execution Case Time
WCS	Worst Case Space
SNR	Signal-Noise Ratio
BER	Bit Error Rate
OS	Operating System
CPU	Central Processing Unit
OS	Operating System

CTP	Collective Tree Protocol
RPL	Low Noisy Networks
VDHL	Very High Speed Integrated Circuits
ISO	International Organization for Standardization
USB	Universal Serial Bus
FIFO	First In, First out

1. Introduction

1.1 Motivation

Wireless Sensor Networks (WSNs) consists of tiny electronic devices that communicate wirelessly and are typically deployed in a field in large numbers to monitor specific phenomena, such as temperature, humidity, and soil moisture). WSNs have multiple applications in many areas, such as agriculture, military, localization, health, and environmental. For example, sensor networks can be deployed in a vineyard to monitor the temperature of the vineyard in order to mitigate for crop damage [1] [2] [3] [4]. Also, WSNs can be used to detect foreign chemical agents in the air or in the soil [5]. They have been used to monitor volcanic activity by deploying multiple sensors on the top of a volcano to consistently measure the temperature and the vapour of the volcano [6]. Basically, WSNs are pervasive in this day and age.

WSNs nodes are distributed across the operation field in huge number in order to monitor the phenomenon with a good resolution. For that reason, WSN nodes are difficult to reach once the nodes are deployed on the operating field. This challenge illustrates the necessity of testing and verifying the software system prior to deployment. In addition, the current design and analysis of software systems for WSNs occurs primarily at the coding layer, which prevents the design from going by the typical software development cycle and correct for errors early in the design stages. Implementation of the design at the coding layer, leads to a decrease in code portability and to platform-specific implementations [7].

The process of developing WSN software code is very challenging since the software design is prone for errors and the debugging process is very time consuming. Moreover, the limitation of user interface I/O hardware of each node (only 3 LEDs as output devices), leads difficulty to test the software while the software is operating and as a consequence, debugging

and correcting the system software code is very challenging. If the code errors are not detected during the implementation and verification stages of development then they may appear once the system is deployed and is operational, which is difficult and challenging to correct due to the operational conditions of WSN systems.

WSNs often consist of self-powered nodes that have limited power resources, which impose design constraints focused on the improvement of power consumption. For example, there is a requirement to improve the power consumption due to the limitation of the power resources otherwise the node the die and will not operate if the node ran out of power. The power consumption of each node is strongly influenced by the software design, the timing constrains of the software modules, the routing protocols used, and the network topology. Design decisions that do not take these issues into account can result in a battery drain on the sensor node, which eventually will lead to disabling the sensor network.

Software modeling enables the designer to capture the design of the software system at higher abstraction layers than at the coding layer, therefore, the designer is able to test and verify the design before the actual deployment. Also, representing the software design at higher abstraction layers gives the chance for early detection of any errors and, as such, early correction of the errors. Additionally, by using various tools of model execution techniques and code instrumentation, the designer can predict the power consumption of the software system. Moreover, by leveraging model driven approaches, the designer can generate and deploy the software code and consequently save the time consumed to implement and debug the code. Also, software modeling enables the designer to design the software system independent of the platform.

1.2 Thesis Statement and Scope of Research

1.2.1 Problem Statement

In general, the development of WSN applications occurs almost entirely at the coding layer, which results in an inability to take advantage of the software development life cycle, and reduces the chances of early detection of errors, verification for design requirements, and design improvements for power consumption before the deployment of the code as stated by Losia et. Al. [7] and Sharma et. Al. [8]. The resulting implementation is often time consuming to develop and debug, platform specific, prone to errors, and is developed during the last stage of the design. There is a lack of software modeling and analysis techniques for system operational requirements and power consumption for WSNs at an abstraction layer other than the coding layer.

1.2.2 Thesis Statement

WSN modeling allows for early detection and verification of the system design performance than the current analysis at the coding layer. Using modeling executions techniques and instrumentation methods, this thesis provides an approach to analyze the system requirements and power consumption characterizes for the sensor nodes components in a WSN. Moreover, this thesis integrates the WSN simulation tools with the UML software models to analyze the power consumed during the communication processes, and feedbacks the simulator results to the design modeling layer.

1.3 Thesis Methodology and Contribution

1.3.1 Thesis Methodology

This thesis focuses on providing WSN developers the opportunity to develop a model for the software in a sensor node that is part of a WSN and the ability to analyze the model for

against any power design requirements. Furthermore, the WSN sensor node software design can be improved based on the feedback results obtained from the analysis of the models and sensor network simulations. This thesis does not focus on developing a design, which guarantees the Quality of Service (QoS) and power management as much as the availability of enhancing the design model based on the analysis results. The steps taken in this thesis methodology were as follows and correspond to Figure 1-1.

1. **Reviewing WSN software modeling approaches.**

As a first step several software modeling approaches for WSN systems were reviewed in order to determine the best modeling language for capturing and analyzing the design of a WSN. Also, the review helped in becoming familiar with the modeling and analysis strategies used at the software modeling layer. This review is presented and discussed in detail in Chapter 3 along with a comparison of those approaches to the approach presented in this thesis.

2. **Representing the sensor node software design at the modeling layer.**

This thesis developed 4 UML design patterns that guide a software designer to capture the design of the software of a sensor node at the modeling layer. The patterns capture the design components and the design behavior using the UML class diagram and state-charts diagrams. IBM Rational Rhapsody tool was leveraged to create the diagrams.

3. **Verifying the design requirements and analyzing the power consumption at the modeling layer.**

This thesis has leveraged the UML model execution technique to validate the design against the requirements and analyze the power consumption for the hardware components, such as LEDs, sensors, and ADC. The validation procedure and the analysis were done by executing the UML model and instrumenting the state-chart diagrams with both the power consumption

annotations and the timing annotations. Based on the validation procedure, the designer is able to correct and improve the design, which is considered as the first feedback analysis to the design. The model patterns and the verification are discussed in chapter 4.

4. **Generating the code and configuration files for the network simulator**

A simulator is used to estimate the power consumed by the communication components, which is not deterministic and hence very challenging to model in the UML modeling layer as were the hardware components. A code generator was developed in order to generate the code and the configuration files required for the WSN network simulator. The code generator parses the XML representation for the UML diagrams that represent the software design of the sensor node. This XML representation is generated by the UML modeling toolkit. Some of the sensor network information does not exist in the design model, such as the battery capacity and network topology, so the code generator has to prompt the designer for such details. The code generator instruments the generated code in order to facilitate integrating the results of the simulator back to the model.

5. **Validating the generated code**

The generated code from the UML model was validated using instrumentation statements that were added by the code generator. The code is executed on the network simulator and it displays the calls between the design's components defined in the UML model. The output from the instrumentation statements are compared to the UML executed model in order to ensure that the generated code workflow expresses exactly the behavior captured in the design. The code generator and validation procedure are discussed in detail in chapter 5.

6. **Power consumption analysis for the communication process**

The simulator simulates the activity of each node in the network and generates a power consumption log file for each node. This thesis developed a parsing tool that uses the simulator's power consumption log file report together with the instrumentation output to display the power consumption at each portion of the UML model. This parsing tool enables the designer to relate the communication process analysis to the model layer and to integrate the simulation results with UML model.

1.3.2 Thesis Contribution

This thesis developed a framework and methodology for model level analysis for WSNs. The framework allows the designer to represent the design at the modeling layer and enables the designer to early detect the design errors, validate the design against the system requirements, and improve the power consumption of the design. The thesis leverages the simulation tools to analyze the networking power consumption and integrates the simulation results with the design representation at the modeling layer (i.e. UML models). The developed framework contains the following developed tools:

- Representation of the WSN design at the modeling layer using UML diagrams: UML patterns are developed to capture the WSN design components and the WSN design behavior. Representation of the WSN design at the UML layer enables the designer to early verify the system requirements at the modeling layer and analyzing the power consumption of the sensor node leveraging the UML model execution feature. The developed framework contains the following developed tools:
- Generating code and the configuration files for a network simulator from the UML model: A code generator is developed in java to parse the XML representation of the

UML model and generate the equivalent code for the network simulator. The code generator also created the configuration file required for the simulator. The code generator also instrument the generated code with debugging statement to ease the integration between the simulation results and the UML models

- Parsing the simulation results tool: A tool has written in java code to parse the simulation results and feedback the results to the model, which enable the design to integrate the simulation results with the model. The integration of power analysis of the simulator is essential to feedback the simulator power analysis of the communication process back to the model.

1.4 Tools leveraged in the thesis

The tools that were used can be categorized into 2 groups:

- The tools that were developed as part of this thesis contribution. The tools developed are the code generator and the parsing tool used to parse the results from the sensor network simulator. Those tools are explained in Chapters 4 and 5, respectively.
- The tools that existed and were leveraged to develop the UML models and analysis of the power consumption related to those software components that displayed deterministic power consumption characteristics. The tools used were the IBM rational Rhapsody for UML models and Avrora WSN simulator. Tools leveraged and developed in this thesis

1.4.1 IBM Rational Rhapsody

IBM Rational Rhapsody [9] is an UML software design environment developed by IBM to ease a software designer on the use of UML tools and model driven approaches in software design. IBM Rational Rhapsody contains many features; however, this section contains the overview of the features that were used in this thesis:

Component Diagrams: Rational Rhapsody offers component diagrams that capture the system components, sub components, interfaces, operations, and attributes . Rhapsody contains a new feature called a composite class, which is a part of the component diagram. The composite class is considered as a container for the sub components and captures the structure of one main component. The composite class offers the feature "class part" which enables one class to create instance of another class. Therefore, one class can signal a method or an event of another class. The class part feature is essential to enable the animation of the sequence and state-chart diagrams, which captures the behavior of the design.

State-Charts: The state-chart defines the behavior of a component or a sub component. The behavior is captured through a group states, transitions, events, and triggered operations. Rational rhapsody offers the state-chart execution feature which executes the model and highlights the current executed state. The events can be triggered through a state within the same state-chart or from another state-machine of a separate component through using the class part feature. This feature enables the designer to specify the interaction between the system sub components and the main components. Additionally, the state-chart execution feature enables to the designer to inject events (manually generate the events) and observe the system behavior in case this particular event has occurred.

In addition, Rational Rhapsody enables the designer to add code to the state-chart to do a specific function so that the code is executed once the state is active. This feature is very helpful to instrument the state-chart which can be used for analysis purposes as explained in chapter 4. Moreover, the state-charts can be annotated to represent the time constrains of the system by timing the transition of the state-chart. The timing constrains facilitate the validation procedure of the requirements as well as the system analysis.

Sequence Diagrams: The sequence diagram presents the interaction between the system components in the form of messages. The diagram contains the component life line, the messages exchanged between the components, and the time different between the messages. The messages on the diagram can represent an event, a triggered operation, or internal method.

The sequence diagram offers two modes:

- *The design mode:* this mode creates methods, operations, and events once the designer draw them in the diagram and they are considered as part of the design.
- *The analysis mode*: this mode enables the designer to verify the behavior of the design and validate whether or not the design meets the functional and the non-functional requirements.
    - The designer can draw the messages, time constrains, events, actors on a sequence diagram and save it as the non-executed sequence diagram. The messages, and events drawn have no effect on the design.
    - The designer runs the animation mode of the design and the sequence diagram will capture the actual trace of the system in an executed sequence diagram.
    - Rational Rhapsody offers a tool for comparing the sequence diagrams. The comparison verifies if the executed sequence diagram (from the actual system trace) is identical or different from the non-executed sequence diagram. The comparison shows the differences between the two sequences diagrams and test the timing of the signals between the system components, the sequence of the signals, the missing states, etc. The comparison is very helpful tool to validate the system design against the system requirements.

Code Generation: The code generation tool has support to generate the code for four languages; Java, C, C#, and C++. Additionally, IBM rational rhapsody contains a plugin to generate the XML representation for the model. The XML contains representation for developed diagrams. The XML representation helps in the code generation process in case the programming language of the target platform is not supported by Rhapsody.

1.4.2 Avrora Simulator

Avrora simulator is cycle accurate instruction-level simulator (emulator) and is presented in [10]. The simulator is built in java language, so the simulation for the WSN design components takes place by the object oriented principle. Avrora simulator requires; the code representation for the software design, the network topology, and the simulation flags that configure the simulation.

Avrora represents each node of the network with an object and each object runs by an individual thread. By the threads the nodes interact with each other in the terms of sending/receiving messages based on the nesC code of the node. All the nodes are synchronized by the use of one global clock. The node activity is simulated by using a java event queue. In reality the low power consumption profile of OS puts the nodes into sleep where less power is consumed. The node wakes up whenever an event is signaled from a hardware component, clock, or a message received. Consequently, the simulator fills the event queue with the events that are about to be signaled in an order of which the software calls them. Then, the simulator consumes each event in the queue in a first-in-first out (FIFO) ordering.

The simulator has a class to simulate the behavior of the *CC2420* Radio Frequency transceiver chip, which is utilized in many sensor nodes based on the 802.15.4 communications protocol. The *CC2420* simulator class controls the sending and receiving process

of each node through using the *wait()* function so that the program guarantees the synchronization between the sender and receiver nodes. The collision avoidance is considered in the radio program as well. The CC2420 simulator class calculates the summation of all RSSI of all the signals received by the node. If the RSSI is too high, this implies that there are too many nodes sending information to the receiver node and thus the radio program will force the sender node to wait or re-send the message later. Also, the RSSI value can be used to determine the range and location discovery of sensor nodes. Moreover, the program that simulates the radio communication process of the nodes simulates the 802.15.4 Link Quality Indicator (LQI). LQI consider the signal-to noise ratio and the measurements of the RSSI value in order to find the best parent node for each node in topology. The RSSI between the two nodes is determined through a topology configuration file, which is configured by the designer of the sensor network. The topology configuration file contains the X Y Z coordinates for the layout of the sensor nodes and the density of obstacles (if not present is zero) for each radio link. Based on the physical distance between the nodes, the simulator estimates the RSSI between the nodes.

Avrora offers instrumentation tools for debugging and analysis purposes. In order to use those tools the designer needs to configure the simulator to reflect the interest to use those tools. There is variety of tools offered by the simulator; however we will highlight the tools that are used in this thesis only.

Debugging Statements in Avrora: The debugging statements in Avrora are capable of printing any statements while executing a specific location of the code as well as the time/cycle of the statement execution. The debugging statements do not require any changes in the simulator structure.

Calls Monitor in Avrora: The monitor call is a debugging tool that is supported by Avrora in order to monitor the calls between the node components and the interrupt handlers. In this thesis approach, monitoring the call was leveraged to validate the generated code by comparing the monitored calls with the equivalent calls at the modeling layer.

Power Consumption Monitoring in Avrora Tool [11]***:*** Avrora has a power model for the components of the nodes; the model was developed based on real time measurements and code executions of the sensor node components. The model covers the power consumption for the LEDs, CPU, Board, and Radio. The power consumption tool has the capability to generate a power consumption log file for each node in the network. The power consumption tool updates the log file whenever the power profile changes. The update contains the cycle number when the update occurred and the amount of power consumed for each component. Also, the power consumption tool can predict the life time of each single sensor node. At the end of the simulation, the tool prints out for each node; the node life time and the power consumed for each hardware resource in the sensor node.

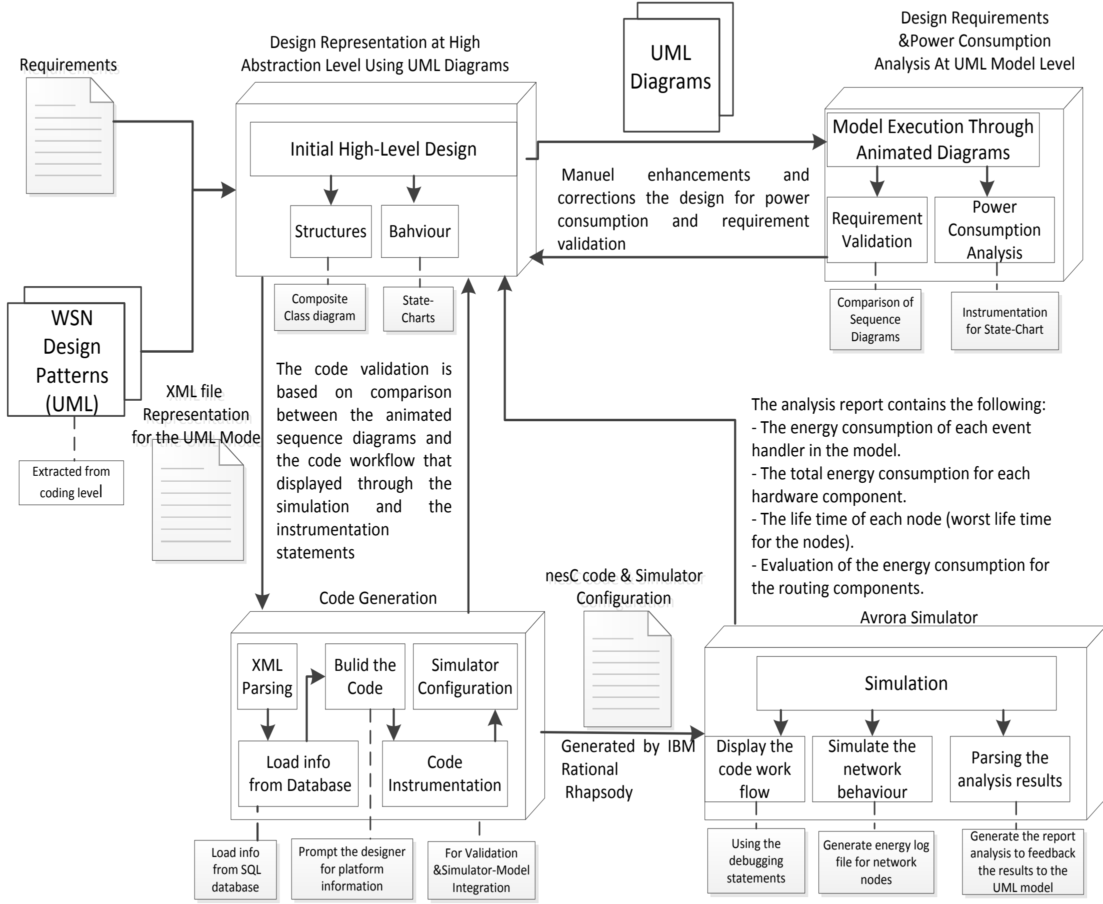


Figure 1-1: Thesis Methodology Diagram

1.5 Thesis Structure

The structure of this thesis is as follows.

- ***Chapter 2*** contains the background material and contribution of this thesis. This chapter discusses the problem statement of this thesis and the methodology that is followed to solve this problem. The methodology explains the tools that are used and the tools that are implemented to solve the problem as well as the requirements at each design stage so that the reader gets a good overview about this thesis objective and this thesis contribution.

- ***Chapter 3*** discusses the prior art in software modeling for wireless sensor networks and compares this prior art to the approach taken in this thesis.
- ***Chapter 4*** explains the UML design patterns that were developed as a part of this thesis contribution, as well as, the proposed methodology in using these design patterns to validate the operational system requirements. Chapter 4 also presents an approach to calculate the power consumption of any deterministic components in a sensor node by the instrumentation of the software model entities. In order to analyze the power consumed due to the communications caused by the sensor network, it was necessary to integrate the UML models with a sensor network simulator.
- ***Chapter 5*** explains the code generator tool developed that parses the UML model and generates the equivalent code for the network simulator. This chapter also presents the validation procedure used against the generated code by instrumenting the generated code using debugging statement. The debugging statements ease comparing the software behavior at the coding level to the software behavior at the model level. Moreover, this chapter contains examples of generation and validation procedure of the code in order to demonstrate that the algorithms work properly.
- ***Chapter 6*** describes the integration between the UML models and the analysis coming back from the network simulator so that alterations to the software design can be performed as opposed to code alterations.
- ***Chapter*** *7* presents a complete study cases that demonstrate how to design the sensor system at the modeling layer, analyze the power consumption for the deterministic components, generate the code for the simulator, and finally simulate the design and

integrate the simulation results back to the UML models. Chapter 8 contains the conclusions and future work.

- ***Chapter 8*** contain the thesis contribution, the limitation of the developed methodology, and the future work.

2. Background

2.1 WSN Structure

Wireless Sensor Networks (WSNs) consists of tiny electronic devices that communicate wirelessly and are typically deployed in a field in large numbers to monitor specific phenomena, such as temperature, humidity, and soil moisture). The nodes communicate wirelessly in order to forward the packets to one gateway where the information is further processed.

Typically, each sensor node contains a wireless unit to communicate with the other nodes, a processor unit, one or more sensors attached to the node and Analog to Digital Converter (ADC) unit. The sensors sense the phenomenon and generate the analog signal that represents the sensed data then transfers the analog digital to the ADC that converts the sensed information to a digital form. The sensor node loads the digital information into data frames and sends the data by the wireless unit to the parent node. The nodes communicate with each other wirelessly to transfer the sensed data to one gathering node, which is known as a collector node. The collector node could be attached to a gateway to facilitate the transfer of data between the WSN and other devices on more conventional networks like those based on the Internet Protocol (IP).

Most of wireless nodes that are used in a WSN have a range of 500 meters. Therefore, the nodes rely on the multi-hop communication routing protocol to deliver the information from the source node to the destination node, especially if the WSN monitors the phenomena in a large field. The task of the routing protocol is to build the data path by establishing the routing tree structure. This structure defines the data path for each node by defining its parent node. The most common WSN routing protocols (i.e. CTP [26] and RPL [28]) use the expected transmission count (i.e. ETX) to estimate the link quality between each node and its neighbor

node. The ETX is a ratio between the successfully delivered frames and the total number of attempts to send those frames. Therefore, the lower the ETX is the better link quality the nodes have. Each node selects the parent node based on the lower ETX. Using this policy will avoid the infinite loops of forwarding the data packets. The root node (in most cases the gateway) broadcasts a frame which indicates that its ETX is zero so that all frames are collected in the gateway. The tree structure is maintained by broadcasting beacon frames to check the availability of the parent nodes. In case one of the nodes fails, the routing protocol builds a different path to overcome the broken link.

The tree structure dynamically changes based on the availability of the nodes and the packet delivery rate, which is determined by the ETX. Also, the packet delivery rate is affected by the environmental conditions of the transmission process. All of these conditions lead to the dynamic changes of the tree structure and the taken path by each data packet. Therefore, the communication process has a non-deterministic nature.

The data packet wakes up the node from the low power listening mode and signals the radio to start receiving the packet. Once the receiving process is completed, the radio component signals an event to indicate the completion of the process. The behaviour of the nodes when the packet is received can be different from one design to another based on the implementation of the event handler. The designs can be classified into the following categories:

- Partially non-deterministic design: The non-deterministic portion of the design is the receiving and forwarding of the received frames from the child nodes due to the dynamic changes of the typology. Therefore, the implementation of the receive event handler contains a simple code to forward the data packet until the packet

reaches the gateway. The rest of the node activities, such as sensing, controlling the ADC unit, and controlling the LEDs is fully deterministic since they are activated based on a specific timer and signalled for a specific rate of time.

- Deterministic design: Some of the designs are fully deterministic where the communication process occurs based on a pre-defined point-to-point communication node.

2.2 WSN Power Consumption

The power resources of WSN nodes are very limited, since each node is powered by rechargeable batteries. Therefore, the design of any application that executes on a sensor node must to be essentially power efficient to avoid the node failure. Therefore, the power consumption requirement is crucial since the life time of the node relies on the life time of the battery. Therefore, power consumption improvement is essential to maintain the life time of the node, especially during the dark hours since the batteries are not being charged. The power consumed by the node can be classified as follows:

- ***Sensor Node Power Consumption:*** This power consumed by the sensor node hardware components, such as (LEDs, ADC, Sensors, etc.). The amount of power consumed is proportional to the rate of signaling those components. The software design controls the rate of signaling those components and the duration of operating each component.
- ***Sensor Network Power Consumption:*** This power consumption by the sending and receiving process that occurred during the communication between the nodes. This portion is not consistent among all nodes and among all operational conditions due to the unstable topology nature, as explained in the previous section, and due to the

unpredictable data control packets. The communication process consumes around 60% of the total power consumption of each node.

There are multiple factors that influence the power consumption of each node, such as the following:

- The routing protocol that is used to control the communication process.
- The data collection methodology since the data collection methodology controls the number of packets sent and received.
- The software structure that controls the signaling of the hardware components.

2.3 WSN Modeling

The design of WSN systems usually occurs at the implementation level and does not involve design at higher levels of abstraction. This leads to a decrease in code portability and to platform-specific implementations [2]. A WSN system produced using this approach is prone to both design and implementation errors and is very challenging to debug (user interfaces to sensor nodes are very limited so even simple text output is challenging). If errors are not detected during the implementation and verification stages of development, then they may appear once the system is deployed and is operational.

Modeling of a WSN enables the designer to capture the design at higher abstraction layers before the actual implementation of the application. This facilitates fixing and correcting design errors using diverse methods of design analysis. Early model analysis for the design enables the user to evaluate the design before the actual deployment. Moreover, modeling enables the designer to display the flow of the design so that the designer is capable of capturing and enhancing the design before the actual code implementation. Also, integration of the power

consumption analysis and the system model facilitates the evaluation of the design energy consumption.

The characteristics of modeling techniques can be categorized as follows:

- ***Language Syntax***: the modeling language syntax can vary between graphical (i.e. UML) or textual syntax such XML language.
- ***Language Scope:*** the language scope can vary based on capturing either the procedural activity of the node or capturing structure of the node.
- ***Language Tools:*** the tools associated with each language can vary based on the aim of the modeling. For instance, some of the modeling language support model execution, other support code generation.

The system details that are captured with each modeling language are strongly dependent on the aim of the modeling, while the rest of the system details are abstracted out. An extensive survey was conducted for the existing modeling language for WSN and was explained in Chapter 3.

3. Modeling Techniques for Wireless Sensor Networks

3.1 Introduction

The objective of WSN software modeling is to represent the WSN design at a higher abstraction layer than at the coding layer. The result of this is to facilitate analysis of the WSN prior to any significant coding. The design details that are captured at the modeling layer are strongly dependent on the objective of modeling while the rest of the details are abstracted out, such as code generation, or analysis network performance. This chapter presents the results of a survey of nine modeling techniques for WSNs. The modeling techniques are different from each other in the terms of notation, the objective for the sensor model, the representation of the sensor node architecture, and the representation of the system architecture. Some of the modeling techniques capture the sensor node architecture, such as the sensor node software, and the interaction between the node's software components. Other modeling techniques capture the WSN system architecture, such as the interaction between the nodes and the topology structure. In this chapter, each modeling technique is explained briefly by showing the objective of the modeling, the modeling technique notation, the representation for sensor node software architecture, and the representation for the WSN architecture. Also, the software modeling approach that was developed in this thesis, was presented in order to compare this thesis approach to the art in software modeling techniques for WSNs.

The rest of the chapter is organized as follows; Section 3.2 gives an overview of the reviewed modeling techniques, Section 3.3 discusses the modeling of the node architecture, Section 3.3 presents the modeling of system architecture, and Section 3.5 has the conclusion.

3.2 Modeling Techniques Overview

This section provides an overview of each of the modeling technique included in the chapter. Some of the modeling techniques use an appropriate notation to support the aim of the modeling, such as UML, while others use their own notation, such as the Insense technique [12]. The techniques use different basic elements (e.g. channels, processes, modules, components) to express a WSN as a model. A channel is used to represent the communication between two elements of a WSN. For example, channels can represent the characteristics of sensor-node communication, node-node communication, and node-gateway communication. Processes, modules, and components are used to represent the sensors and nodes of a WSN. Also, the modeling techniques vary with the tools used by the modeling technique to achieve the modeling purpose. For example, some techniques use code generation tool to produce the operating system code, other modeling technique use model execution tools to analyze the software model. The modeling techniques surveyed also vary in terms of the scope of modeling. For example, some techniques are intended to model a single sensor node; some are intended to model the complete WSN.

3.2.1 High-Level SDL Models (HL-SDL)

HL-SDL is a modeling language that uses the Specification and Description Language (SDL), which is normally used to model and simulate communication protocols as defined in [13]. Dietterle, et al. [14] adapted SDL to analyze the worst case execution time (WECT) of the node design and to generate the required nesC code for TinyOS once the analysis is completed. The HL-SDL uses SDL processes (i.e. extended finite state machines to model TinyOS components. The node is modeled as a collection of channels and processes. The HL-SDL approach focuses on modeling the node behaviors and did not capture the system

architecture. Dietterle, et al. has mentioned that the generated code requires a manual improvement in order to operate properly. An example of manual improvement of WSN generated source code is modifying the communication between the components from asynchronous in the model to synchronous in the target platform. In addition to manual improvement of the generated source code, simulation can be used at the model level to refine the model (with respect to performance) prior to code generation.

3.2.2 Insense

Dearle et Al. [12] developed the Insense modeling language, which is java based language to create a component-based model for a WSN. Insense is built and run on the Contiki operating system, which is a popular operating system used for WSN [15]. Insense models the node behavior and analyzes the component behavior for worst case execution times (WCETs) and worst case space (WCS) within a given WSN node. The components in Insense model the behavior of the software and the hardware component. An example of a hardware-software binding is the interaction between the sensor (e.g. humidity, temperature or moisture) and the software component that handles the readings. The sensor type is modeled as a component that uses a communication channel to transfer data to the software components [9].

The model components are concurrent and they communicate synchronously via directional channels that are used to abstract away from low-level synchronization and communication issues. To the best of our knowledge, the approach does not have support to model the WSN system architecture.

3.2.3 MathWorks Modeling Approach

Mozumudar et al. [16] have developed a framework in MatLab environment that aims to design, simulate, and generate the code for WSNs. The node behavior is modeled as a

parameterized state flow block. Nodes in the MathWorks approach also contain timing and random number generators that are used for simulation. MathWorks leveraged the tools such as state-charts execution, chart displays, scopes, and plots, to perform the algorithmic analysis. According to the analysis results the model can be refined. The final stage is to generate the WSN code using the Target Language Complier (TLC) which can generate C code for MANTIS [17] and nesC code for TinyOS. The Mathworks analysis approach has also been used successfully to generate the code for Power Efficient and Reliable In-Network Aggregation (EERINA) [18]

In MathWorks, the framework is able to model the static topology by modeling the nodes with state chart and the communication medium which is implemented in C, models the connectivity between the network nodes.

3.2.4 Model Driven Engineering Approach (MDEA)

Losilla, et al. [7] used UML and a Model Driven Engineering (MDE) approach that includes three modeling layers:

- ***WSN Domain Specific Modeling***: a meta-model that created by a domain expert.
- ***Component-based Platform Independent Models (PIMs):*** A UML-like language primarily composed of activity diagrams and state-machine diagrams.
- ***NesC Platform Specific Model (PSM):*** used with the UML PIMs to generate the NesC source code.

Transformation rules were defined that control moving from one modeling layer to another. Moreover, manual refinement can occur after every transformation to improve the generated model. The MDEA approach is supported by the Eclipse IDE as well as a number of Eclipse plug-ins (e.g. MOFScript) that are responsible for automating the transformation process.

The MDEA approach is used to generate the nesC code for the MITRA WSN application, which was designed for the application domain of precision agriculture. For the system architecture, MDEA does not have representation for the topology structure. However, the MDEA approach develops a UML model for the gateway node and a model for the network nodes, assuming that all the nodes have the same behaviour.

3.2.5 Promela and the SPIN Model Checker for WSNs (PM)

V. Oleshchuk proposed using Promela and Spin model checking technique to check the WSN network connectivity. The technique uses Promela language as an input language for the design model. The approach included the physical location of all the dynamic nodes and supports adding and removing nodes as well as changing their physical location. The Spin model checker is used to perform a network connectivity check. Specifically, the physical location data of the nodes is analyzed in conjunction with the data about the coverage range of the sensors [19]. The only technique that did not model node behavior was the paper using the Promela Model Checker. The authors of this work decided to simply focus on the modeling of network connectivity as opposed to including any significant modeling of the node behaviors.

3.2.6 SensorML

SensorML approach is an XML based language that supports modeling each sensor by specifying the sensor’s meta-data (e.g. sensor ID, sensor type) as presented in the SensorML specification document manual [20]. The model includes representations of the physical elements (e.g. sensors and actuators) and the non-physical element (e.g. mathematical operation within the sensor). All of the elements are modeled as processes that are linked together explicitly by inputs and outputs. Linked sequences of processes form process chains that correspond to the behavior inside a single node [4]. SensorML has support to model network

elements, such as base station, sensors, and the network topology. SensorML does not provide any sort of model execution techniques that enables the designer to improve or test the design before the actual deployment.

3.2.7 SystemC-AMS

M. Vasilevski et. Al in [21] have developed SystemC-AMS that analyzes the communication channel between two nodes for Bit Error Rate (BER) and Signal –to Noise Ratio (SNR). The approach combines C++ with block diagrams to model the WSN and simulate the system. For each node SystemC-AMS models the Analog to Digital Converter (ADC), the microprocessor, and the communication channel between the node and the next hop node. On the system architecture level, the approach models the communication process between two nodes. However, the modeling technique does not have support for the network topology since the approach assumes that data is transmitted by single hop communication.

3.2.8 UM-RTCOM Model

M. Diaz et. Al. [22] developed the UM-RTCOM modeling technique, which is a real-time component based modeling framework written in CORBA and developed to analyze the design for Wireless Sensor Actor Network systems (WSANs). WSANs have the same functionality as WSN; however, WSAN also reacts in response to the sensed data. The approach analyzes the design for WCET, deadlock freedom, and verification of live time of the design nodes. The modeling technique assumes the network is composed of sensors that have identical functionality, and actors of the same rules. The modeling approach captures the system architecture through modeling the network elements (sensors, actors, and the gateway) as three virtual machines (VMs), where each VM models a single element. The system behavior is modeled by the interaction between the three VMs. Sensors communicate

with actors and actor's communicate with the coordinator over channels . Communication via a channel is modeled as a tuple. A tuple is a sequence of fields with the form: *(t1 , t2, ..., tn*) where each field ti can be: a TC identifier (or) a value of any established data type of the host language where the model is integrated. The communication channel protocol modeled in UM-RTCOM has been tested in an actual sensor network deployment by Barbaran et al. [23]. The deployment shows the improvement of the middleware overhead compared to another deployment where the motes send the sensed data periodically to the actors.

3.2.9 eXtended Reactive Modules (XRM)

Demaille, et al. [24] have extended Reactive Modules (RMs) language to develop XRM language. The authors have used WSN design as a case study to demonstrate the capability of the developed modeling language. The approach leverages the model execution technique to calculate the package delivery probability and the power consumption of the nodes. The technique uses a module that captures the behavior of each single node in the network. Each module captures the node behavior, the communication capability, memory, and power consumption. XRM models the power consumption by deducting the power consumed value from a local variable for every time an activity is provoked in the module. The model has multiple modules to model the network nodes. Each module contains X-Y variable to indicate the physical coordinates of each node. Also, the modeling technique supports executing the modules which models the behavior of both the node and the system behavior.

3.3 Overview from the Node's Software Architecture Perspective

This section discusses how the software modeling techniques capture the software architecture of the elements in a sensor node, such as nodes, sensors, actuators,

software components, and hardware components. In particular, we consider the modeling of the node structure and the node behavior.

Most of the reviewed modeling techniques used a form of component-based modeling to represent the software in a sensor node. The WSN behavior is modeled by specifying the component's internal behavior, component to component interactions, and the communication channel's characteristics. The approaches reviewed can be divided into two distinct types. Those that focuses on the augmentation of the models to capture particular features such as concurrency, event-driven behavior, and real-time behavior and those that leverage standard models like state space and procedural coding that were later used for code generation or performance analysis.

Most of the reviewed modeling techniques reviewed used to create a platform independent model for the node software architecture. However, there is a necessity to include some of the hardware details for the following reasons: most of the modeling techniques surveyed can be used to create a platform independent model for the node architecture. However, there is a necessity to include some of the hardware details for the following reasons:

- The software behavior is tightly coupled to the hardware elements of the node. Therefore the binding of software and hardware components should be represented in the model. For example, some of the sensor boards require signaling the sensors sequentially and not concurrently. Therefore the hardware properties impose a specific behavior on the software.
- The hardware information may be needed to be represented in order to be able to generate source code from the model. Generated source code is interacting with

the node hardware (timers, ports, sensor types) and therefore the model has to be aware of the hardware components in order to generate the correct code.

- The design analysis tools require some information about the hardware specification to develop the proper analysis results. For example, the WECT that is calculated by some modeling techniques is highly dependent on the hardware response time. Also, calculating the total power consumption is strongly related to the hardware used in the designing process .

3.4 Overview from the System's Architecture Perspective

The surveyed modeling techniques capture the WSN system architecture by representing network behavior, the topology structure, and the network behavior. Modeling network architecture is crucial because many important performance values are based on the network:

- Analyzing the network power consumption requires the representation of the network behavior since the communication process (sending and receiving) consumes most amount of power.
- Analyzing the packet lose requires the representation for transferring the data packet across the network.
- Analyzing the schedule-ability factors, such as WECT and WCS requires captures the time responses of the network elements.
- Analyzing the network connectivity requires capturing the network topology structure.
- Analyzing the communication channel between the channels requires the representation of the communication environment.

The topology of WSN systems can be dynamic or static. The static topology represents the nodes in a fixed location while the dynamic topology represents the nodes while they are in a

moving state. Additionally, based on the modeling target, the technique models the number of hops in the network design. We can categorize the topology representation by the modeling techniques into two categorize;

- ***Group Representation***: For some the Modeling technique, representing the physical location is not essential to their approach, however, there is a necessity to represent various elements, such as the gateway, nodes, and actuator. Therefore, each element is represented with a separate model components. For instance, UM-RTCOM has a virtual machine to capture all the nodes behavior and another virtual machine to capture the actuators. MDEA uses a UML model to captures the design of the gateway and another one to represent all nodes.
- ***Single Representation***: Other modeling techniques the physical location of the node is very essential to their approach so the model contains a representation for each single element in the network. Each node is defined by the X-Y coordinates.

3.5 Model and Simulation Integration

The modeling techniques for WSNs are capable of representing the design at higher abstraction layers and are capable of developing the system analysis at the modeling layer. Some of the parameters that the modeling techniques can analyze the design for are SNR, power consumption, WECT, WCT, and packet lose rate as explained in Section 3.4.

Some of the modeling techniques [25] [26] [27] have integrated the simulation tools such as OMNeT, Matlab, and Pmodel with the system model in order to develop further analysis for the system, such as testing the design performance, system scalability, and system optimization. The integration between the model and the simulator requires a coupling tool to intermediate the model domain and the simulator domain and requires a group of defined rules

that controls the transformation process between the model and simulation requirements. The coupling tool ensures that the underlying coding language, which is required by the simulator, matches the model semantics.

The integration between the model and the simulation tools enable the designer to assess the designer for various QoS parameter, such as the reliability, integrity, safety, and power consumption. Based on the survey that was published in [28] for 33 modeling techniques, the integration between the models and the simulation tool requires the generation of the simulator configuration. The generation process for the configuration faces a lot of challenges, such as the flaws in the generated items, as well as, the quality of the generated items. Moreover, based on the classification for surveyed techniques, some work should be invested in the feedback simulation results to the model, especially for the analysis of the non-functional requirements.

The feedback of the simulation analysis to the model enables the designer to evaluate the QoS parameters of the design at early stages of the design cycle. In addition, the feedback allows the designer to conduct the analysis which is very challenging in the model level. The feedback should be mapped to the model using the model terms so that the designer can relate the analysis results to the model. Additionally, based on the survey results, the best way to present the feedback is to visualize the system trace that is analyzed by the simulation and compared the feedback to the system trace that is displayed in the sequence diagram.

3.6 Concluding Remarks for the Review

Modeling technique enables the designer to represent the design at higher abstraction layers, which ease testing and correcting the design before the actual deployment. The elements can vary to be process, modules, components, diagrams, or channels. Modeling helps to resolve

some of the WSN software implementation challenges by the following (see Table 3-1):

- Calculating the system performance parameters, such SNR, BER, WECT, WCS, or power consumption.
- Generating the deployment code so that amount of effort used to write and debug the code is saved
- Checking the design correctness.

Approach	Notation	Modeling Scope	Modeling Elements	Modeling Purpose
HL-SDL [14]	SDL	Node	Process-channels	nesC code generation + model execution to analyze (WECT)
Insense [12]	Insense Language	Node	components, channels	Analysis for WECT and WCS
Mathworks [16]	State Diagram and C	Node-System	State-charts, communication medium	NesC and C code generation tool + functional analysis
MDEA [29]	UML	Node-System	Components (wireless link as class)	nesC Code generation
PM [19]	Pormela	Network	Processes, channel	Model checking to verify the connectivity of the nodes
SensorML [30]	XML- Source Code	Node	Components-processes model	
SystemC-AMS [21]	Block Diagram – C++	Node	Block diagram, source code	Model Execution. Design analysis for SNR and BER
UM-RTCOM [22]	CORBA	Node-System	Components, channels	Analyze WCET, enhancing the communication properties
XRM [24]	eXtended Reactive Modules	Node-System	Modules	Analyzing the Packet delivery probability and

				power consumption

Table 3-1: Overview of the Modeling Techniques

The scopes of the modeling technique can various to capture the node architecture, the network architecture, or both architectures. Modeling techniques uses different modeling elements to capture either the node behavior or the network behavior. The modeling techniques are targeted to analyze specific software challenges like concurrency, real-time, and event modeling. Modeling at the system level is also another feature that some modelers support.

As seen in Table 3-2 several modeling techniques like UM-RTCOM, XRM, PM, MDEA, and MathWorks can all model the sensor network but there is a focus for each on what behavior they model. They all model node activity and take into account node to node communication but not all can explicitly model the network topology as is the case with MDEA.

Approach	**Node Behavior**	**Network Behavior**	**Network Topology**
HL-SDL [14]	Concurrency, event-driven	-	-
Insense [12]	Concurrency, real-time	-	-
Mathworks [16]	Procedural, state space	Node/base station interaction	Single hope. Static topology
MDEA [29]	Procedural, state space	Node/base station interaction	-
PM [19]	-	Node connectivity	Multi hop, dynamic topology
SensorML [30]	Event driven	-	-
SystemC-AMS [21]	Procedural	-	Single hop, static topology
UM-RTCOM [22]	Concurrency, real-time, event-driven	Nodes/actor/base station interaction	Single hop, static topology
XRM [24]	Procedural, state space	Power management-wake up states	Single hop, static topology

Table 3-2: Modeling of Node and System Arcitecture

This thesis approach has the following similarities to the modeling approach:

- Representation the topology as X-Y coordinates of the nodes using the strategy that is introduced by XRM.
- Calculating the power consumption using the strategy that is introduced by XRM and HL-SDL.
- Leveraging the state-chart to capture the node behavior, this is a similar to MDEA approach.
- The model of the node is platform independent. The platform specifications are added by the code generation, which is a similar approach to MDEA and Mathworks.
- Using the model execution techniques to analyze the system performance parameters.

However, none of the modeling techniques have presented a design patterns that are able to capture WSN design semantics. Additionally, our approach is capable of validating the functional represents of the design early during the design stages.

Additionally, this thesis introduced a methodology to validate the generated code by leveraging the sequence diagram execution feature and simulation results. Also the generated code by our approach does not require any sort of manual improvement, which is the case of MDEA and HL-SDL approach.

None of the modeling techniques have introduced a mythology to improve the design based on the analysis results. However, we have implemented a parsing tool to analyze the simulation results and integrate the results with the UML model so that the designer is capable to improve the design at the modeling layer.

4. UML Modeling and Power Consumption Analysis for Wireless Sensor Networks

4.1 Introduction

This chapter introduces a group of UML design patterns, which enable the system designer to capture the design at a higher abstraction layer other than the coding layer. The introduced UML design patterns allow the designer to capture the design requirements, determine the design OS support components, define the design structure through using class diagrams, define the design behavior through using state-chart diagrams. Capturing the design using the UML diagrams gives the designer the opportunity to verify the design requirements, and to analyze the power consumption through the UML model execution technique and the comparing sequence diagram feature. Also, the chapter introduces the instrumentation for the state-chart that eases the analysis process. Based on the verification and the power consumption analysis, the designer is able correct and improve the design.

The UML patterns were developed from the code of a typical data acquisition sensor network designed for the acquisition of temperature values in a vineyard [1]. This system was coined SensIV and was developed by leveraging the TinyOS operating system. The SensIV system integrates multiple system design aspects typical of collector style WSNs, such as sensing the phenomena, transmitting the data to a collector node, controlling system hardware elements (i.e. LEDs, ADC, and transducers), having a static defined location of the nodes, and leveraging some dynamic tree collecting routing protocol like the Collector Tree Protocol (CTP) [31].

The database provides the code generator with the TinyOS component information, such as the component nesC syntax, the required interfaces of the component, the events nesC

syntax, and the variables associated with each event. The database structure contains one parent table and four child tables.

Figure 4-1: Modeling, Verification, and Analysis at the Model Layer

The chapter is structured as follows: Section 4.2 explains the architecture of the WSN design and the required components; Section 4.3 introduces the UML design patterns with examples, Section 4.4 demonstrates the validation procedure for the system requirements; Section 4.5 explains the methodology that was used for design verification; Section 4.6 contains the related work; and Section 4.6 presents the summary of the chapter.

4.2 WSN Systems Design Architecture

Wireless Sensor Networks consist of several nodes deployed in a field to monitor specific phenomena, such as temperature and humidity. The resources that are used in the majority WSNs system are as follows [32]:

- ***Transducers***: The transducers sense the phenomena and generate the analog signal that represents the sensed values. The transducers are known as sensors as well. The

nodes can be connected to multiple sensors of the same type or different types. Therefore, one node can sense one phenomenon only or multiple phenomena.

- ***Analog-to-Digital (ADC):*** The ADC converts the analog signal generated by the sensors to a digital form so that the radio unit loads the digital signal into a payload frame and sends the data to the next node.
- ***CPU*:** The CPU controls the activity of the node. Normally, the CPU alternates between the sleeping, idle, and active modes based on the power profile of the design. For example, the low power profile forces the CPU to be in the sleeping mode while the node does not execute any activity.
- ***Radio Unit*:** The radio unit is responsible for sending and receiving the routing protocol control packets and the sensor data to and from the next node.
- ***Battery*:** Each node contains rechargeable batteries. The batteries are charged during the day by using a solar cell connected to the node. The batteries are considered as limited power resource, and therefore, the system needs to be power efficient due to the difficulty of reaching the nodes after deployment.
- ***The Operating System*:** WSN operating systems offer a group of tools to ease the design process of the system. WSN operating systems are event driven operating systems, such as TinyOS [32] and Contiki [15] OS. Each operating system offers a group of components, which are denoted as OS hardware and software components.
    - The OS hardware components contain a group of parameters, events producers, and methods that manages the signalling between the hardware resources and the application software. For example, an ADC hardware

module that contains the methods that turns on the ADC unit and then turns off the ADC unit once the conversion process is completed.

  - The OS software components contain a group of parameters, event producers, and methods that ease implementation of the functionalities required for the system, such as routing procedure. For example, TinyOS offers two routing protocols: Collective Tree Protocol (CTP) [31] and Routing Protocol for Low Power and Low Noisy Networks (RPL) [33]. The software components offers the methods to start the routing components in order to start passing the routing message, send the data packets, and stop the routing components. Once the component is started, the routing component takes the responsibility to build and maintains the routing tree.

The design process for WSN systems involves various stages based on the system requirements. The typical design process of a WSN system involves the following stages:

- Define the application requirements.
- Define the design structure that determines the software and the hardware components that are necessary to build the system.
- Define the design behaviour that is controlled by the deployed application software. The application software controls the hardware and software resources by the signals between the design components, the timing, and the signal sequences between the components. The designer defines the behaviour of the software so that the system requirements are met.
- Validate the system design against the system requirements.

- Analyze and calculate the power consumption of the design based on the design behaviour.

4.3 Design Patterns for Wireless Sensor Networks

The design patterns introduced in this section were developed based on TinyOS programming manual. The programming manual quoted the definition of the pattern as "*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"* [34]. The programming manual introduced a group of TinyOS coding patterns as well as the description of the TinyOS nesC code structure. UML design patterns were developed from some of the coding patterns that were defined in the programming manual and were applied to SensIV system. The defined UML design patterns enable the designer to capture the design behavior and the design structure at the UML modeling layer.

Each pattern is introduced by illustrating the pattern intent, which explains the purpose of developing the pattern; illustrating the pattern motivation, which explains the general design feature that the pattern captures; illustrating the pattern structure, which explains the UML diagrams structure that represents the patterns, and illustrating an example that explains how to apply the pattern. The template used to explain the design patterns follows the pattern explanation introduced in [34].

4.3.1 Sensor Node Service Component Design Pattern

4.3.1.1 Pattern Intent

The pattern captures the software and hardware components at the UML layer, which the designer uses to build the design structure.

4.3.1.2 Pattern Motivation

At an early design stage, the design structure is built by selecting the design hardware and software resources and selecting the proper software and hardware components to interact with these resources. Each component offers the service to the application software so that the application software can access the equivalent resources. The component contains a group of parameters, parameterized methods, and event producers to ease the access of the resource. The application software can create one or many instances of the service component. For example, the application can create four instances of a sensor hardware module to access four sensors attached to a sensor node.

4.3.1.3 Pattern Structure

The hardware and the software components are captured in the UML layer by defining a group of stereotype classes. The UML stereotypes classes contain a representation of the parameters, the methods, and the event producers captured by a class diagram and a state-chart to capture the behaviour of the stereotype class. The developed stereotype classes are as follows:

- ***Sensor Stereotype Class***: This class captures the implementation of the sensor hardware component that controls the sensor hardware. The class has the methods that trigger the sensors and produce the event with a parameter that has the sensed value.
- ***ADC Stereotype Class***: This class captures the implementation of the ADC hardware component that controls the ADC unit. The class has the methods that turn on and off the ADC.
- ***Radio Stereotype Class***: This class captures the implementation of the radio hardware components that controls the radio unit.

- ***Routing Stereotype Class***: The routing software component is used to setup each data packet path from the source node to the destination node. The application software starts and controls this routing component. The UML routing class represents the routing component at the UML layer methods and the class contains the method that controls the routing process of the data packets.

4.3.1.4 Sensor Node Service Component Component Example

An example of the behavior of the sensor hardware component is shown in Figure 4-2. The state-chart contains the signaled operation called *sensor1_on and sensor2_on,* which captures the behaviour, *read* that starts the sensors and offered by OS support component. Once the method *sensor_on1* is called, the state-chart changes the state from *Idle* to *ADC_On* in order to model the action of sensing. The state-chart stays in this state for 250ms and then signals the event *readDone*. The time parameter of 250ms is calculated based on real time measurements for the sensor behavior in [35]. This state-chart also captures the sensor component's capability to turn the sensor's ADC on and off.

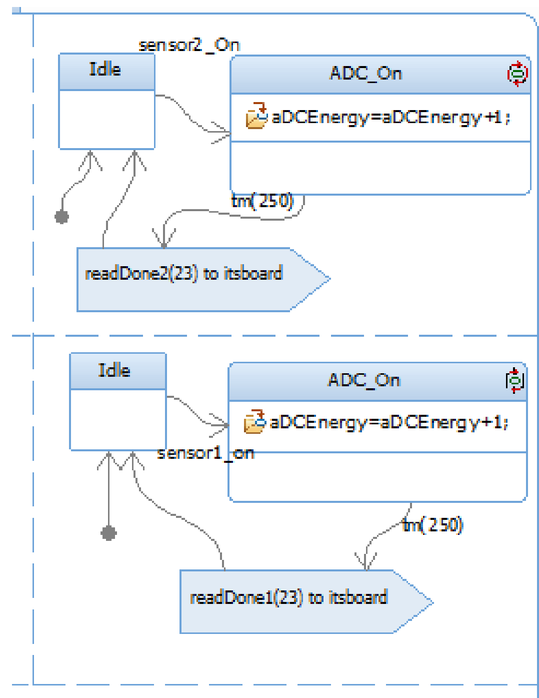


Figure 4-2: State-chart of Sensor Hardware Module

4.3.2 Sensor Node Event Handler Design Pattern

4.3.2.1 Pattern Intent

The pattern captures the multiple events and event handlers structure of the software.

4.3.2.2 Pattern Motivation

Each node in a WSN network contains multiple hardware components, such as ADC, sensors, LEDs, and radio. The node software communicates with the hardware components to start or stop the hardware resource and to pass and return the processed data. For instance, the software signals the sensors to starting sensing, and then the sensors return the values back to the software once the sensor completes the process. The software signals the radio hardware component and passes the sensed data to the radio to be sent out. The synchronization of the communication process between the software and the hardware resources blocks the node's activity until the hardware completes the task and releases the resources. Meantime, the

CPU cycles and the memory resources are wasted. According to TinyOS programming guide, using the synchronized communication is recommended as long as the hardware operation does not operate for a long time.

Alternatively, the designer can use multithreading to manage multiple hardware resources and avoid wasting CPU cycles. However, a WSN is considered as an example of an embedded system, which has limited memory resources (i.e. limited RAM). Therefore, using multithreading on handling the hardware tasks is not recommended since the memory resources are very limited (i.e. TinyOS memory is 512 bytes).

TinyOS offers another approach called the split phase communication approach. This approach achieves bi-directional communication between the hardware and the software components. The software can call the command, which signals the hardware to accomplish a specific operation. The hardware will start executing the operation, while the software executes the rest of the flow without waiting for the hardware to complete the execution. Once the hardware completes the execution, the hardware issues an interrupt for the software by signaling an event to indicate the completion of the operation. The software can call multiple commands to multiple hardware components since the software components are not blocked while waiting for the hardware to complete the operation. Therefore, TinyOS starts to schedule the multiple commands using the first-in-first-out (FIFO) technique.

For example, the software signals the sensors to start sensing by calling the command *mysensor.read().* The software is not blocked while waiting for the sensor to complete the process. Once the sensor completes the operation, the sensor signals an event to indicate that the operation is completed. In the event handler, the software signals the radio hardware resource to start sending the information (see Figure 4-3)

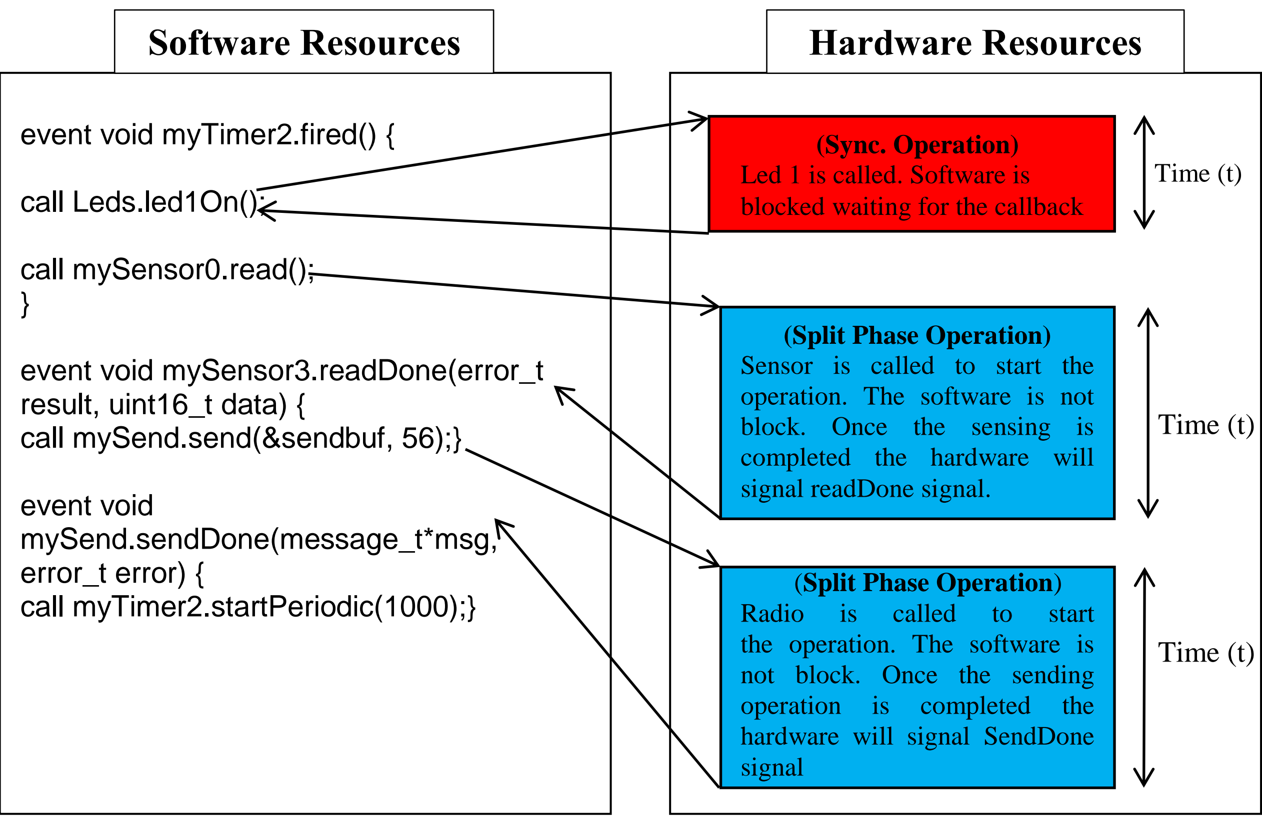


Figure 4-3: Split Phase Communication

Some typical events which are used in WSNs are signaled by the following components:

- ***Timers:*** Timers generally control the flow of the entire application. For instance, once the sensing timer is fired, the application calls the sensors to read the temperature.
- ***Sensors:*** Once the sensor finishes reading data values an event is triggered.
- ***Routing Protocol:*** The routing protocol signals an event once the node starts to send the routing messages to the neighbor nodes.
- ***Sending Unit:*** Once the sending component finishes sending the data, an event is signaled to indicate that the data is sent.

4.3.2.3 Pattern Structure

The event based nature of an embedded OS leads to the possibility of signaling any event at any instance based on the process computation time. The same approach is used to capture the

event in UML. The events are signals by the timers of one of the stereotype classes. The event translates the state-chart from the *Idle* state to a main state. The main state contains sub-states, which capture the steps of the event handler. After the state is executed, the system returns to *Idle* and waits for the event to be signalled again.

The event based nature of an embedded OS leads to the possibility of signalling any event at any instance based on the hardware computation time. In order to capture this behaviour, the *AND* state was used. The *AND* state runs the sub-states concurrently. Therefore, the application state-chart is ready at any time to be signalled by any event captured in the state machine.

4.3.2.4 Example

A portion of the state-chart that manages the sensors is shown in Figure 4-4. The state-chart of SensIV contains the following event handlers:

- ***The event readDone0:*** This event is signaled by OS support component that manages the sensors, once the first sensor completes the sensing process. The event handlers contain the instructions to turn the LEDs on, load the array with the returned value, and finally signal the second sensor to start sensing.
- ***The event readDone1:*** This event is signaled by the hardware component once the second sensor finishes sensing as well. For this activity the same procedures are executed as those presented for the event readDone0.

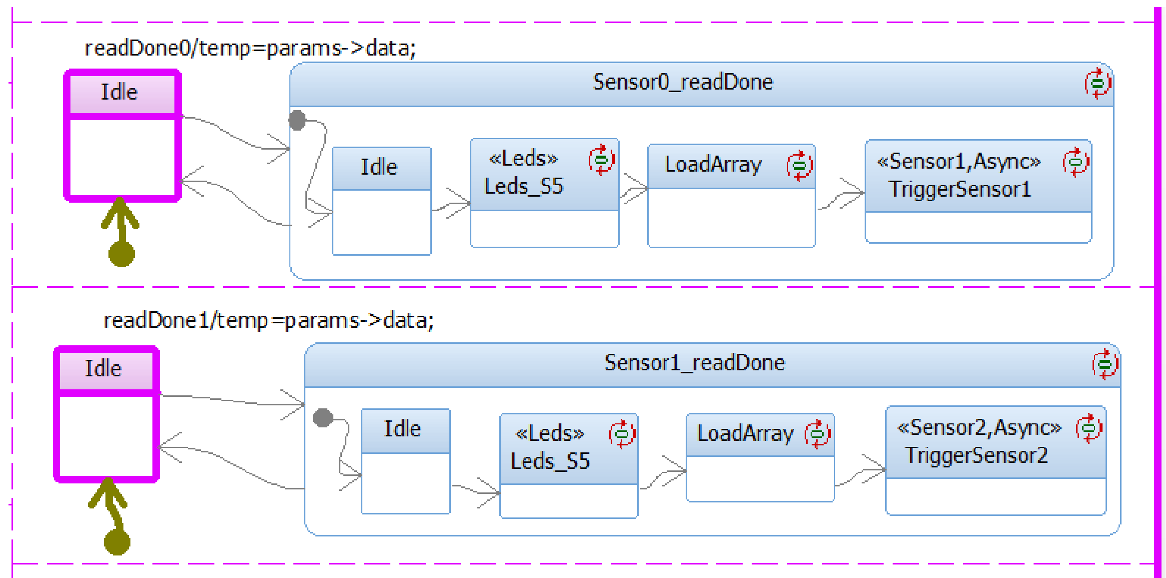


Figure 4-4: Portion State-Chart for SensIV System

4.3.3 Association of the Application Software and OS Support Components

4.3.3.1 Pattern Intent

The pattern captures the association between the application software OS support components.

4.3.3.2 Pattern Motivation

As mentioned in the previous section, each operating system for a WSN offers a group of components that ease accessing the hardware resources of the node and offer software services to the application. This pattern captures the association between these OS components and the application. Each platform has its own platform specific form of defining the association of the components. For example in TinyOS, the association is specified by a concept known as "wiring" and captured with the following nesC code:

```
myApp.RadioControl -> ActiveMessageC
```

The statement associates the application component *RadioControl* to the TinyOS interface *ActiveMessageC*. *ActiveMessageC* is a TinyOS module, which interacts with the link layer of the

radio unit. Therefore, the application component *RadioControl* has access to the methods and the event handles which were developed for the wireless transmitter.

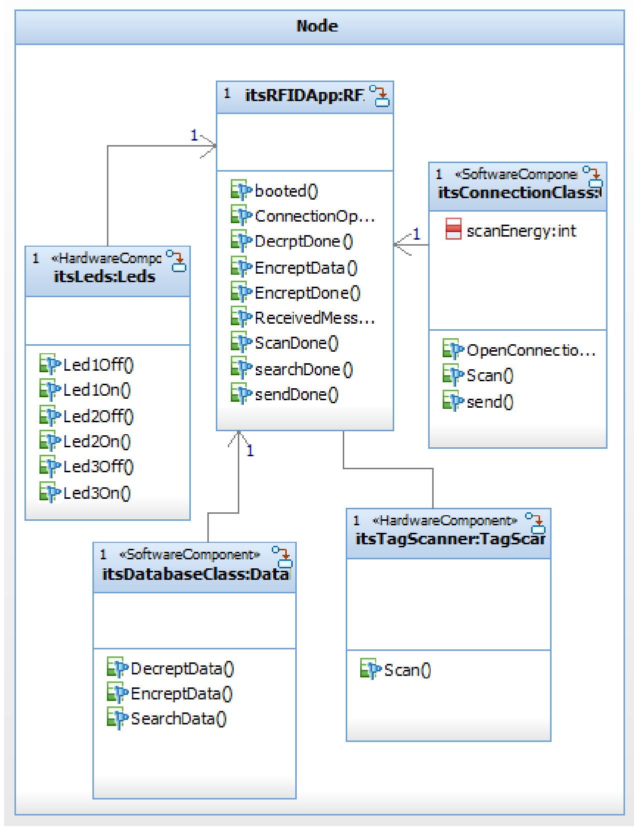
4.3.3.3 Pattern Structure

The *Composite Class* [9] captures the internal system objects, the associations between the system classes, and the collaboration of the interconnected elements of the modeled system. The *Composite Class* supports the creation of a *Class Part* that is supported by Rational Rhapsody [9]. The *Class Part* is an instance of a class in another class and is used to capture the interface between two classes, which are associated to each other in the *Composite Class*. When the *Class Part* is created, the *Class Part* name is change based on the target class name.

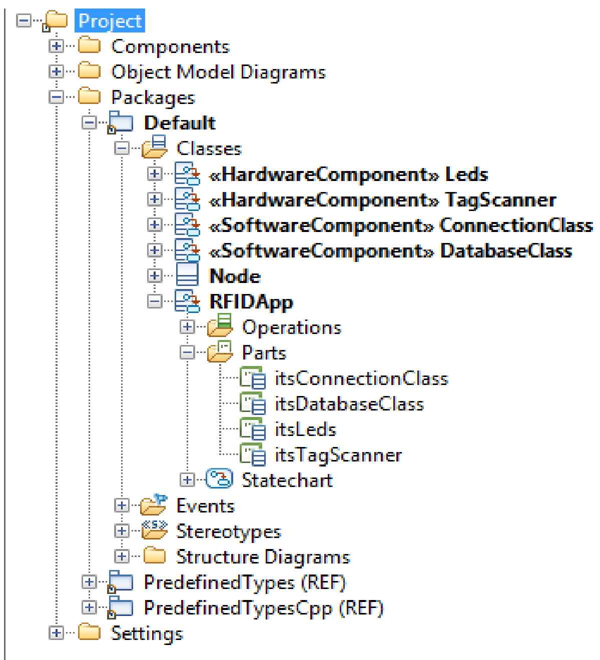
A *Composite Class* is used to capture the instance of a node's hardware components, software components, and the associations between these components to the application, which reflects the tinyOS component wiring design pattern. Moreover, the *Class Part* is required when one of the classes signals one of the events in the other classes. This feature is essential in order to execute the state-charts in Rhapsody that are used for analyzing power consumption of the deterministic components in a sensor node (presented in more detail in Chapter 5).

4.3.3.4 Example

The RFID composite class is shown in Figure 4-5a. The system has one application class, which is called RFID, and three components, which are the LEDs, the database, and the connection. The composite classes display all the components of the system and the association between them by using association arrows. The class parts are displayed in the Rhapsody project panel (see Figure 4-5b). The RFID is an example of point to point sensor networks systems and is used to authenticate the RFID tags. The UML design of the RFID is explained in details in Chapter 6.



A) Composite Class



B) RFID Panel

Figure 4-5: RFID System Components

4.4 State-Chart Stereotypes

As mentioned in the previous sections, the WSN design was captured by a group of stereotypes classes that captured the OS support components, together with the application software component that captured that application's behaviour. The application's behaviour was modeled as a state-chart that contains a group of events and event handlers as mention in Section 4.3.2.

As introduced in Chapter 2, the power consumption analysis of some of WSN System requires running a simulator to complete the analysis, which requires generating the code for simulator. The code generator requires information about the OS support components that are signalled by the application software component. Another group of stereotypes are introduced to the model so that the stereotypes will be an indication for the generator to load the proper information of the components while generating the code based on the target platform component (see Table 4-1). Table 4-1 contains the model stereotype's syntax, an example of the state-chart, contains an explanation of the state and the stereotypes, and contains the equivalent TinyOS component syntax. The stereotype's syntax is generic so that the designer can use the same model for generating the proper code for multiple WSN platforms.

As shown in the table below, each state has the component stereotype and either an Async or a Sync stereotype in order to indicate the synchronization type of the task. The synchronization type of the task is essential to the code generator in order to generate the proper debugging statements that are essential to integrate the simulation results with the UML model as explained in Chapter 6.

Model Stereotype	**Example**	**Details**	**TinyOS Component (nesC syntax)**
Timer	«Timer» StartTimerS1	The state signals the timer component	*TimerMillic*
Sensor	«Sensor2,Async» TriggerSensor2	The state signals the second sensor	*SensorMDA300CA*
ADC	«ADC» TurnOffADC	The timer turns off the ADC unit	*MicaBusC*
Routing	«Routing,Async» StartRouting	The state starts the routing component to send/receive data packets to the neighbour nodes	*CollectionC*
Radio	«Radio,Async» StartRadio	The state starts the radio component so that the radio listens to the network and is ready to send/receive messages	*CollectionSenderC*
Leds	«Leds» Leds_S1	The state controls the LEDs of the node	*LedsC*

Table 4-1: Model Stereotypes

4.5 Analysis in the UML Model Layer

The previous sections have explained the UML model patterns and the stereotypes that were developed to capture the software design and the OS support components of the sensor node. This section demonstrates the capability of verifying the requirements of the design at the modeling layer, in particular the requirements that are related to the power consumption of the system.

The steps in the analysis are listed below:

- The instrumentation of the state-charts.
- The model execution that occurs by using the Rational Rhapsody state-chart execution feature and the sequence diagram comparison feature, which are introduced in Chapter 2.

4.5.1 Instrumentation of State-Charts for Power and Timing Annotation

The behaviour of the stereotype classes that captures the OS support components are captured by a state-chart using the Sensor Node Service Component Design Pattern (see Section 4.3.1). The instrumentation takes place by annotating the power consumption values, as well as, the time response of the operation that is modeled by the appropriate state.

The power consumption annotations are inserted as C++ code instead each state. As long as the ADC is turned on and is not turned off by the signal from the application, the total power consumption of the sensor is increasing. The time is controlled by using the transition condition *tm(Time)* function, which forces the state-chart to be triggered every *t* time and therefore every *t* the total power consumption is increased by the annotated value.

For example, power consumption of the ADC unit is 13mA/second. Once the sensor board receives the signal excitation on, the sensing state and the calculate power state is activated. The calculate power state triggers itself by using the transition *tm(1000)* every 1 second and adds to the total power consumption a 13 value. The power consumption will continue increasing until the ADC component receives the signal *excitation_off* (see Figure 4-6). The annotated values are taken from actual power measurements of the sensor nodes as presented by Zheng et Al. [35].

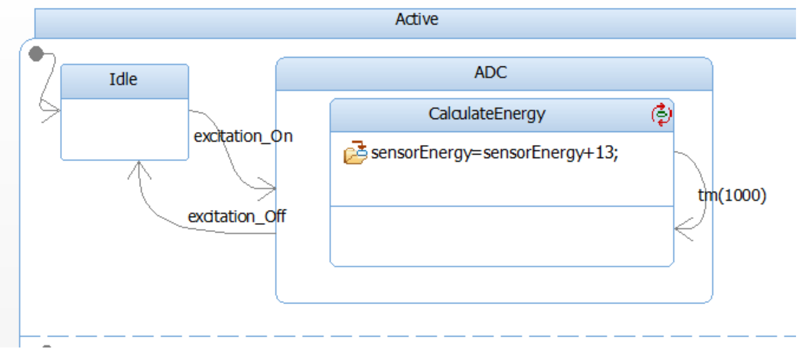


Figure 4-6: ADC State-Chart Model

The Async operations do not require a signal to turn off, such as the sending and sensing operations. Those operations signal an event to indicate the completion of the executed event. Therefore, the power consumption of such operation is calculated once, since they do not require any signal to be turned off. However, the model contains *tm(t)* function to model the time taken by the operation to be executed and signal the event back to the application. This time annotation is essential for the verification of the requirements. For example, the sensing operation consumes 1mA and needs 250ms to complete sensing. The sensing starts once the application signals the sensor with *sensor2_on* event signal and takes 250ms to complete sensing and signals the event *readDone* in the application (see Figure 4-7).

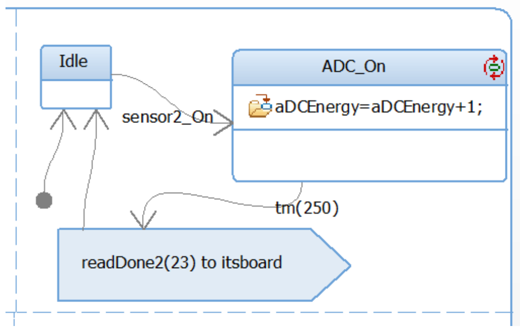


Figure 4-7: Sensing State-Chart Model

4.5.2 Model Execution

The state-chart of the OS support components are instrumented with the timing annotations that are based on real-time measurements. The model execution includes the timing annotations that are inserted in the model. The executed sequence diagrams show the signals that are being exchanged between the design components and the application software, as well as, the timing components.

For example, when the application signals the sensor component to start sending by *TriggerSensor1* state (see Figure 4-8), a signal is sent from the application to the sensor component. Then, the sensor component signals the event *readDone* after 250ms (see Figure 4-7), which models the sensing operation. Once *readDone1* is signaled, the event handler state executes the states *Idle*, *Leds_S5*, *LoadArray*, and *TriggerSensor2* (see Figure 4-8). This interaction between the application and the sensor component is captured in the sequence diagram as well as the timing components. The sequence diagram has life line for each OS component and the software component. On the life line of each component, Rhapsody prints the active state, which the component is executing in the state-chart. Also, the event signaling between the component is displayed in the sequence diagram using arrows (see Figure 4-9).

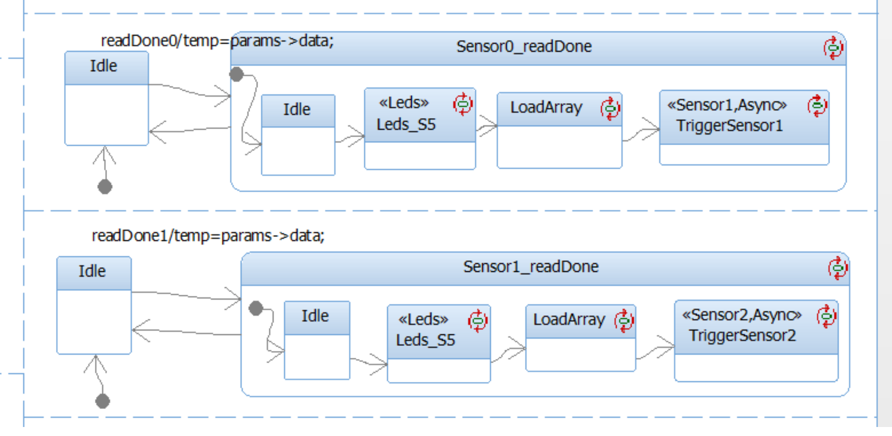


Figure 4-8: Trigger Sensors State-Charts

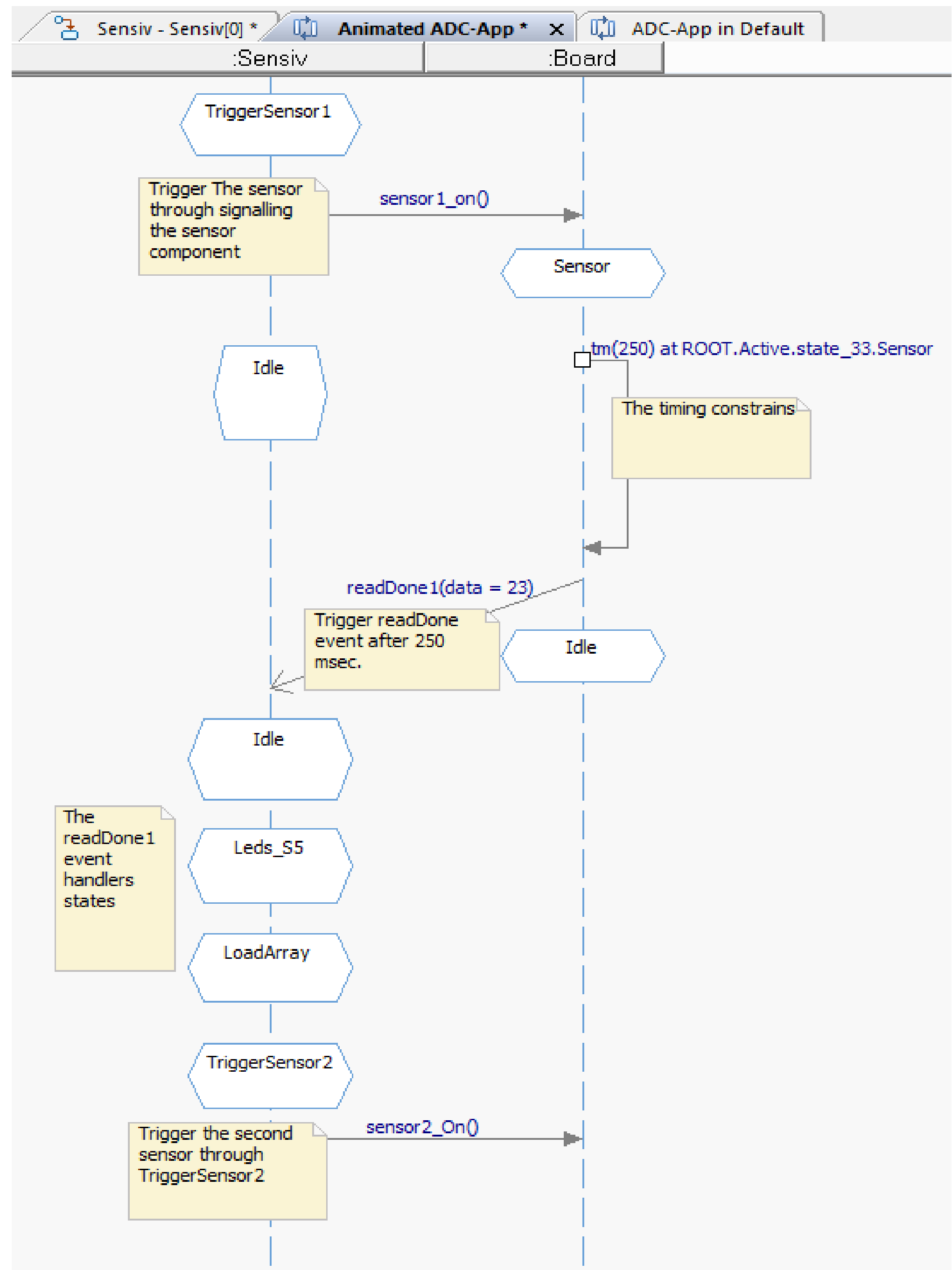


Figure 4-9: Executed Sequence Diagram- Interaction between Application and Sensor Component

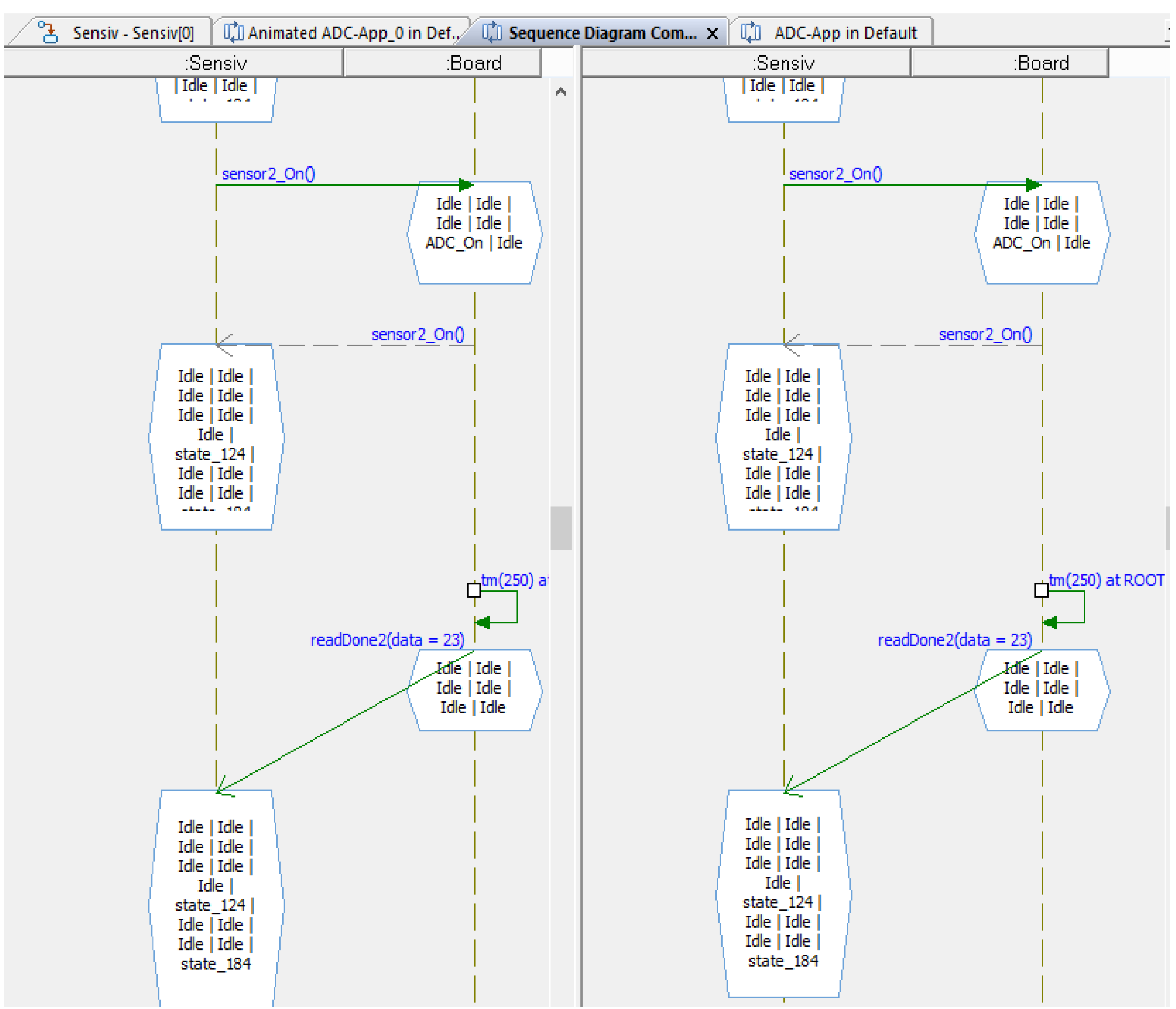
IBM Rational Rhapsody supports the sequence diagram comparison feature. This feature compares the executed sequence diagram to a desired sequence diagram. The desired sequence diagram is created by the designer and contains the expected behavior of the system that meets the design requirements. The sequence diagram comparison feature compares the

following diagram elements: the sequence of the signals, the signals between the components and the application, the timing annotations, the variable types, and the range of the values. The comparison shows if any of the signals between the design components are missing, if any of the signals are out of order, and the timing requirements (i.e. system delays). By using the color code for the sequence diagram elements, the comparison feature displays the matched and unmatched elements between the two sequence diagrams (i.e. the signal sequence, the time components, the variables, etc.).

For example, part of the SensIV application requirements is to trigger the 4 temperature sensors in sequence. When each sensor completes the sensing process within the time range of 250ms, a readDone signal is signaled. In order to illustrate the sequence diagram comparison feature, the trigger signal to the fourth sensor was purposely dropped. Therefore, the executed sequence diagram will only have 3 triggered sensors while the required sequence diagram will have 4 signaled sensors. The comparison results are as follows:

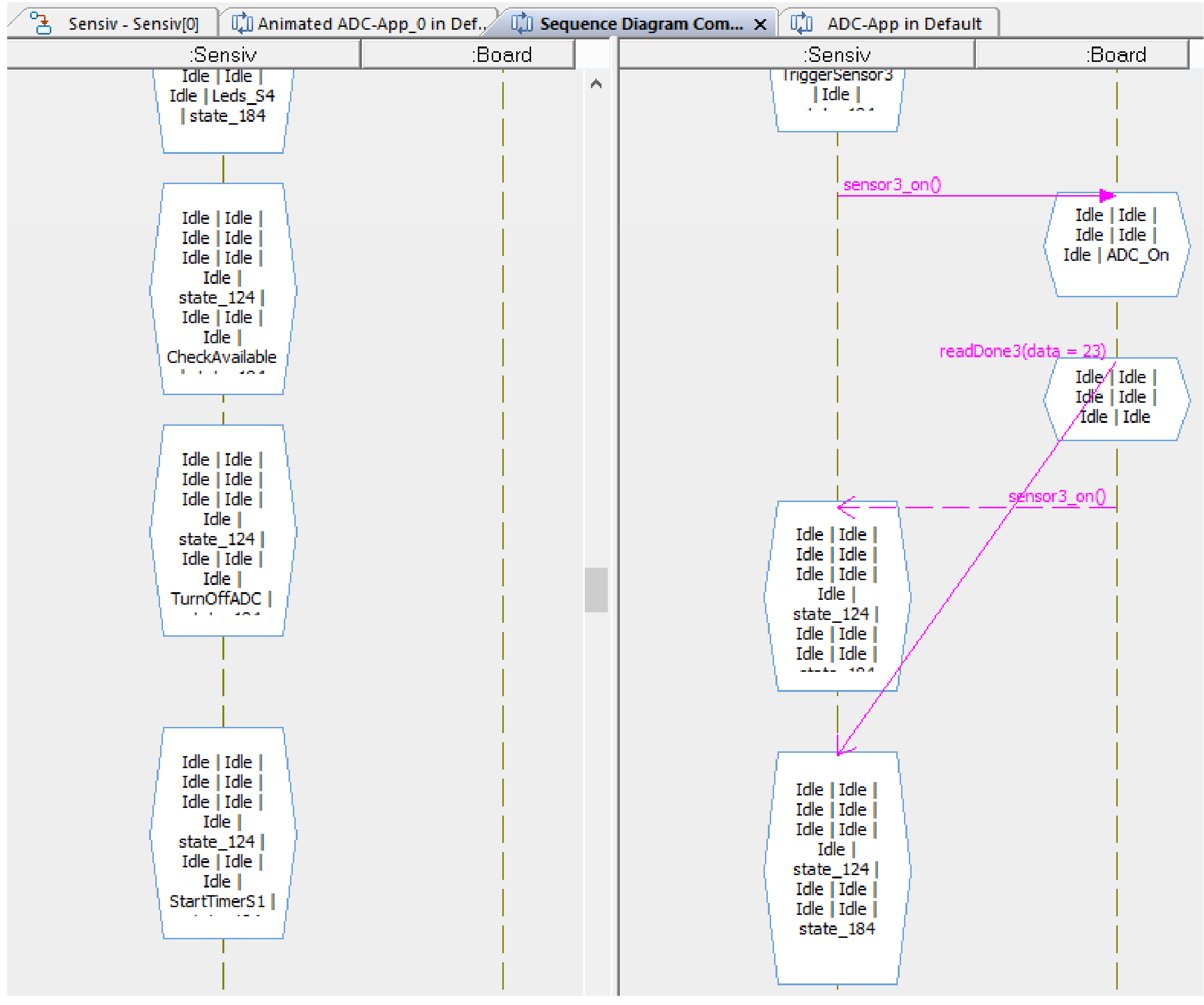
- The signals and the time delay between the components SensIV and the Board of *sensor2_on* operation are matched in both the executed diagram and the desired one. The matched elements between the two diagrams are indicated by using the blue color (see Figure 4 10).
- The signals between the components SensIV and the Board of *sensor3_on* operation are not matched in both the executed diagram and the desired one. The unmatched signals are indicate using the purple color so that the designer can conclude that the design does not meet this particular requirement (see Figure 4-11).

The comparison feature also can be leveraged to visualize the enhancement for the design by comparing the executed sequence of the design before and after the enhancements.



A: Actual sequence Diagram *B: Assumed Sequence Diagram*

Figure 4-10: Sequence Diagram Comparison Results-1



A: Actual sequence Diagram *B: Assumed Sequence Diagram*

Figure 4-11: Sequence Diagrams Comparison Result-2

4.6 Summary and Limitations

This chapter presented three UML modelling patterns used to guide a software designer of WSNs to capture the WSN design architecture and behaviour, as well as to model component interaction in a typical event-based operating system. It also presented an approach of using stereotypes in state diagrams to help analyse the power consumption for the deterministic power components in a WSN by leveraging power consumption and timing annotations in the state charts that are executed at the modelling layer. In order to validate the behaviour of the

WSN, it was demonstrated that the sequence diagram comparison approach could be used. The system architecture was captured by using the association of the pattern of design components while the design behaviour was captured by using the event handler and the service component pattern. The UML patterns leveraged the class diagrams and state-chart diagrams to capture the system in the modeling layer. This chapter also introduced an approach to verify the WSN design requirements. This thesis approach relied on instrumenting the OS support components state-chart with the power consumption values and the timing annotations. The annotated values were calculated based on real-time measurements. In addition, the verification of the requirements of the system was developed by using the sequence diagram comparison feature. The comparison took place between a sequence diagram that contained the expected system behaviour (i.e. assumed sequence diagram) and the executed sequence diagram of the system. Based on the comparison, the designer can evaluate the design whether or not the design meets the functional requirements.

The power consumption methodology that was used in the chapter, calculates the power consumption for the deterministic components (i.e. the LEDs, the ADC, and the sensors). The power consumption for those components is directly proportional to the rate of signalling those components. Therefore, by using the instrumentation of the state-chart diagrams and the executed state-chart diagrams, the power consumption of those components was calculated. The communication process between the network nodes has a non-deterministic nature as explained in Chapter 2. Therefore, the methodology explained in this chapter is not capable of calculating the power consumption of the communication process. WSN simulators have the capability to simulate the network communication process and to calculate power consumption. The WSN simulator requires the code that is deployed in the nodes as one of the required input

for the simulator. The next chapter explains the code generator that parses the XML representation of the design and generates the code for the simulator.

5 Generating TinyOS code for UML High Level Models

5.1 Introduction

The analysis developed in the previous chapter involves power consumption analysis for the hardware resources in a WSN sensor node that exhibit simple direct interactions with the application layer and therefore their power consumption can be deterministically calculated through a power consumption equation. However, in a WSN the communication process between the nodes exhibit a non-deterministic behaviour because the routes that the data packets of each node can vary based on the routing algorithms. In addition, the routing algorithms periodically send routing packets to maintain the topology of the network. The result is that the sending and receiving of messages is unpredictable. Therefore, the Avrora simulator was integrated with the UML modeller to analyze the power consumption for each node, including the power consumption of the communication process. In order to use the Avrora simulator it is necessary to generate executable code from the model that is installed into the simulator. This chapter explains the code generator that produces the platform code and simulator configuration for the WSN Avrora.

The code generator parses the XML representation of the UML model and loads the information required by the platform from a database, such as the code syntax and the OS support components. The code generator prompts the designer for some additional information about the platform that is not represented in the model. Based on the information gathered, the code generator builds the required code and the necessary information for the simulator. In addition, the code generator adds the debugging statements to the generated code, which are necessary to validate the code and to integrate the simulation results to the model as explained in Chapter 6. Currently, the code generator generates nesC code, which is the required

code for the TinyOS operating system. However, the code generator can be extended to support any WSN event based operating system (see Figure 5-1).

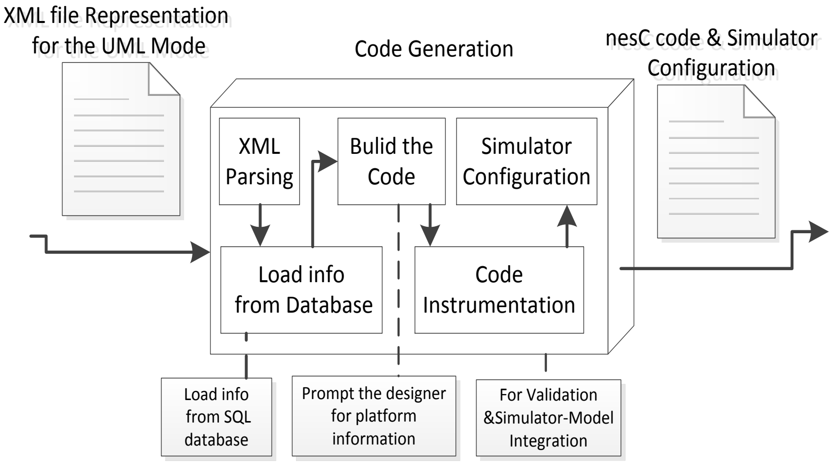


Figure 5-1: Code Generation and Verification

This chapter is structured as follows: Section 5.2 explains nesC code structure; Section 5.3 presents the code generator structure; Section 5.4 discusses the instrumentation of the generated code; Section 5.5 presents examples of the generated code; Section 5.6 demonstrates the evaluation of the generated code; Section 5.7 contains the related work; and Section 5.7 contains the summary and the conclusion.

5.2 TinyOS Code Structure

TinyOS is an event based operating system that is designed for sensor nodes with limited resources (i.e. power, memory, and CPU speed). TinyOS is written in nesC language, which is a C based language. The C code is composed of functions while nesC code is composed of components. Each component offers a particular common service, such as *LedC* component offers the control methods for the LEDs and *CollectionC* component offers the methods, which handles the routing protocol. Furthermore, each component offers a group of specific interfaces, which implement different functionalities of the same component.

The nesC code has 2 modules: the configuration module, which contains the wiring code, and the application module, which contains the application code. The nesC code has the following properties: component wiring, interface declaration, and event handlers. This section explains each property so that the reader can understand the code generation steps.

5.2.1 Component Wiring

TinyOS support components have to be wired to the application in order to offer the service to the application software. The wiring is expressed by explicit statements in the configuration module. The following code lines show an example of the wiring statements:

```
Components CollectionC as myCollectionC
myApp.RoutingControl -> myCollectionC
myapplication.myCTP ->myCollectionC.CTPInfo
```

The first line declares an instance of TinyOS component *CollectionC* called *myCollectionC*. The second line wires the application component *RoutingControl* to the component instance *myCollectionC*. After wiring, the *RoutingControl* component has access to all the methods, events, and parameters in *CollectionC* component.

Some applications require additional services, which are not offered by the component service, but are related to the component service. Those additional services are offered by an upper level component. For instance, the application requires some information about the parent node in the routing path. The routing service is offered by application *CollectionC* component while the routing information (i.e. parent node) is offered by another upper component called *CTPInfo* (see Figure 5-2).

Each TinyOS component contains a specific group of predefined interfaces. Each interface defines a group of functions called commands which the application can call. For instance, the component *CollectionC* has *StdControl, Send, Receive,* and *Packet* interfaces.

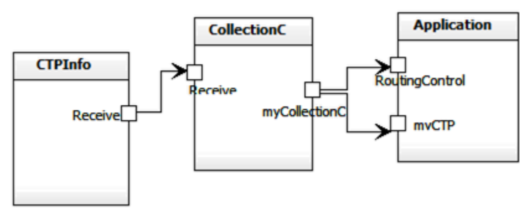


Figure 5-2: TinyOS components

StdControl interface implements the commands *start* and *stop*, which are responsible for starting the routing control packets and stopping the routing control packets, respectively. In the application module, the call for the interface commands is executed by calling the application component which is wired to the TinyOS component followed by the command syntax. For instance, to start the routing control messages:

- The interfaces have to be declared as well as the components. However, the interfaces are declared in the application module. The following code shows the interface declaration for *CollectionC* component.

```
uses interface StdControl as RoutingControl
```

- The declaration for the interface of the *CTP info* component will be as follows:

```
uses interface myCTP as Receive
```

- The interfaces can be bi-directional, either the interface is provided by the component or the interface is consumed by a component. The provided interface has the keyword *provides* while the consumed interface has the keyword *uses*.

5.2.2 Event Handlers

WSN operating systems offer the capability for the system to post a task for the one of the OS support components to be completed. The posted tasks for the processor are scheduled to be executed according to FIFO scheduler protocol. Normally, the tasks are posted for the computational processes, which require a specific period. In most cases, those tasks contain one

or more of the commands which are implemented in the TinyOS support components. Therefore, the command must be called by the appropriate component interface and in the form of a task in order to be processed in the background.

After the command is finished executing, the command triggers an event to indicate that the execution is over. The event is triggered by the TinyOS component interface, which is the wired application component. The designer can implement the event handler based on the design workflow. Some of the events return parameters, which are the output of the execution process. The following code presents an example of posting the task and implementing the event handler:

```
post startrouting();
Task startrouting(){  call RoutingControl.start(); }
// once the routing started:
event void RoutingControl.startDone(error_t error){….}
```

The event *startDone* is implemented in the interface *StdControl* that is offered by *CollectionC* component and wired to the application component *RoutingControl*. The parameter *error* is returned by the event to indicate if the task has been executed properly.

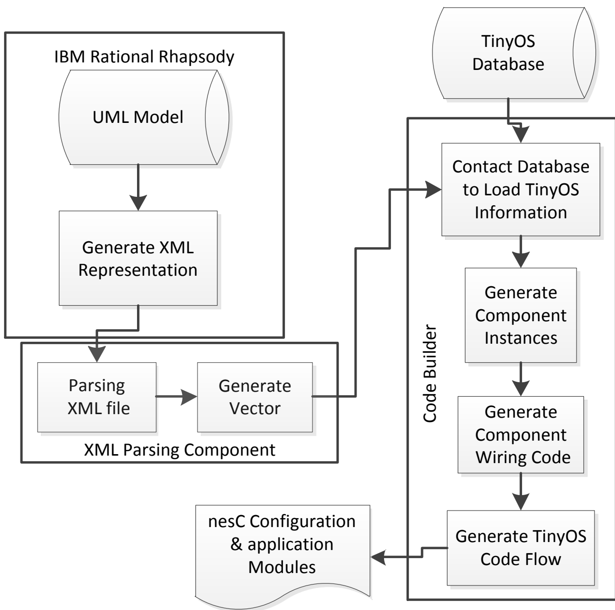


Figure 5-3: Code Generator Structure

5.3 Code Generator Structure

The code generator is written in Java language and contains the the following three main components: XML Parser, Database Structure, and the Code Builder (see Figure 5-3). The target of the code generator is to parse the information provided by state-chart diagram for the design and generate the equivalent nesC code. The code generator has the following functions:

- Parse the XML representation for the UML model.
- Recognize the system components, which are used with the design.
- Contact the database and load the required TinyOS components information.
- Generate the instance of each component.
- Wire the TinyOS support components to the application.
- Declare the component interfaces which are used by the application.
- Generate the application code workflow.

5.3.1 XML Parser

Rational Rhapsody provides a toolkit which generates an XML representation for the UML model. The XML Parser component reads the XML file and extracts the model information. The extracted information from the XML file is the following:

- The event names which exist in the model.
- The main state which is activated when the event is signaled.
- The sub-states inside each main state.
- The stereotype of each state.
- The input and output transitions ID of each state, which is necessary for organizing the code workflow.

There are two packages which the XML Parser uses: *XMLTypes* and *ClassTypes*.

5.3.1.1 XMLTypes Package

An XML file contains lines structured in a tree format, which contains a tag that identifies the type of element the line represents. The lines contain tags for the *statesID*, transitions, stereotypes, etc. The *XMLTypes* package is responsible for storing and extracting this information and saving it in a similar tree structure.

5.3.1.2 ClassTypes

The *ClassTypes* package uses the extracted information by *XMLTypes* and organizes the formation in events, main state, and the sub-states so that they have a similar structure as the nesC code (i.e. event, events handlers, and execution steps in each event handler).

5.3.2 Database Structure

The database provides the code generator with the TinyOS component information, such as the component nesC syntax, the required interfaces of the component, the events nesC syntax, and the variables associated with each event. The database structure contains one parent table and four child tables (see Figure 5-4). The explanation of each table column is shown in Table 5-1.

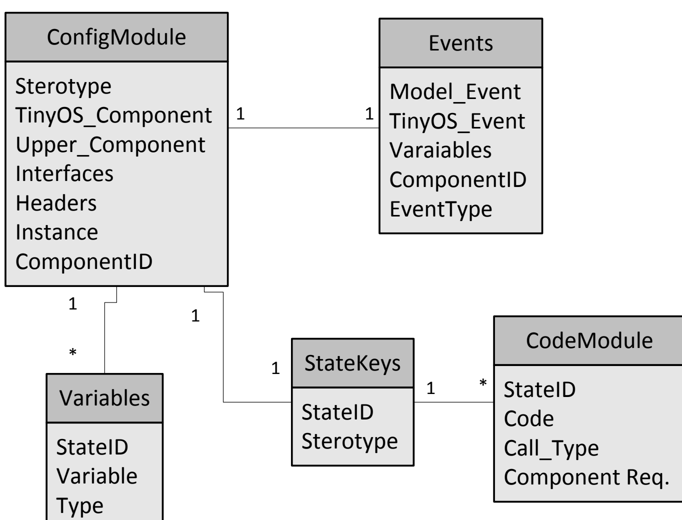


Figure 5-4: Database Structure

Database Table	Column	Description
StateKey	*StateID*	Contains the name of all the states as appeared in the mode (example: *TriggerSensor0*, *Leds_S5,* and *StartTimer1*).
	Stereotype	Contains the high-level stereotypes, which are captured in the model (example: *Sensor, ADC,* and *LEDs*).
ConfigModule	*TinyOS_Component*	Contains the TinyOS support components, which are required for the higher-level components (example: *CollectionC*).
	Upper_Component	Contains the TinyOS support components, which are required for providing service to main component (example: *CTPInfo*).
	Interfaces	Contains the interface syntax for the TinyOS support components (example: *StdControl* for collection component).
	Headers	Contains the TinyOS header files, which are required for the components (example: *Timer.h* that is required for timers)
	Instance	Contains a *True* or *False* value. The true value indicates whether the TinyOS component is initiated.
	ComponentID	Contains the association between the *ConfigModule* table and the *Events* table. The code generator requires information about the component that signals the particular event.
Variables	*Variable*	Contains the variables to be initiated. Some of the states require variables to be initiated (example: *sendbuf* variable)
	Type	Contains the variable type (example: *message_t*)

CodeModule	*Code*	Contains the TinyOS code syntax that is equivalent to the UML state (example: *Leds.led0On()*
	Call_Type	Contains the call type for the command (example: *call* and *post*).
	Component_Required	Contains a Boolean value to indicate whether this code requires the component instance. For example(*led0On()* needs the components association *LedsC*)
Events	*Model_Event*	Contains the event names in the model (example: *startRoutingDone* and *readDone0*).
	TinyOS_Event	Contains the equivalent nesC syntax for the model event name (example: *sendDone* and *booted*).
	Variables	Contains the parameters returned with the triggered event (example: *error*).
	EventType	Contains the return type when an event is called. (Example: *received* event returns a variable of type *message*).

Table 5-1: Explanation of the Database Tables

5.3.3 Code Builder

The code builder is considered as the central component of the code generator. The code builder triggers the XML Parser to parse the XML file. The parser loads the XML file and parses the information and saves it in a form of a java vector. The vector contains multiple objects of the events class. The code builder uses the information in the vector and contacts the database to load the required information. Each state requires wiring code that is generated to the configuration module. Also, each state requires the interface declaration and the code logic that is associated with the state behaviour to be written in the application module.

The configuration module is generated by using the components TinyOS syntax and the upper components syntax, together with TinyOS keywords: *new components, ->, implementation,* and *configuration*.

For the application module, the code generator declares the application module by the declaration of the interfaces using the keywords *uses*, *provides*, and *as*. Also, the event handlers' code is generated by using the keywords *event* and *void* followed by the component name and the parameters, which exist in the information vector and were loaded from the database. The code workflow is generated in the same manner as the configuration and the application module. However, the sub-states in the XML file are out of order, therefore, the code builder checks the in and out transitions unique ID to organize them to generate the proper sequence of code.

For example the state "*Leds_S4*" in the UML layer requires a creating instance line and wiring line in the configuration module *components LedsC*; and *Counter.Leds->LedsC* respectively. Also the "*Leds_S4*" state requires the interface declaration in the application module uses interface Leds as Leds and the code logic lines *call Leds.led0On(), call Leds.led1On(),* and *call Leds.led2Off();*.

As mentioned in the chapter introduction, the code generator's role is to generate the TinyOS support component syntax and the code syntax that are required for the simulator in order to achieve the integration between the UML model and the network simulator. Some of the design specification is not captured in the UML layer of the design. Therefore, before generating the code the generator prompts the designer for such information. For instance, the network topology, the battery capacity, and the platform details since such information is required in the configuration file of the simulator as explained in Chapter 2.

5.4 Instrumentation of the Generated Code

The code generator adds debugging statements to instrument the generated code for two purposes: to help validate the code and to support integrating the simulator results back to the model. The debugging statements consist of nesC print method that are implemented in the nesC header file *AvroraPrint.h*. The print method accepts multiple parameters types, such as string, integer, etc. In addition, the simulator prints the print method parameter once the complier reaches the debugging statement location in the code. Also, the simulator prints the CPU cycle when the debugging statement is executed.

5.4.1 Code validation instrumentation

The validation procedure of the code occurs by comparing the code workflow to the model workflow so that that generated code semantics are verified to match the model semantics, which is known as back-to-back testing [36]. The code generator adds a debugging statement that prints the UML state ID and was translated by the code generator to the nesC statements. Those debugging statements are printed when the simulator executes the generated code and reaches debugging statement location in the code. As shown in Figure 5-5 (A), the debugging statements consist of the nesC function *printStr* and a parameter of type *string.* The parameter indicates the model state ID that caused the code lines to be generated (Figure 5-5-B).

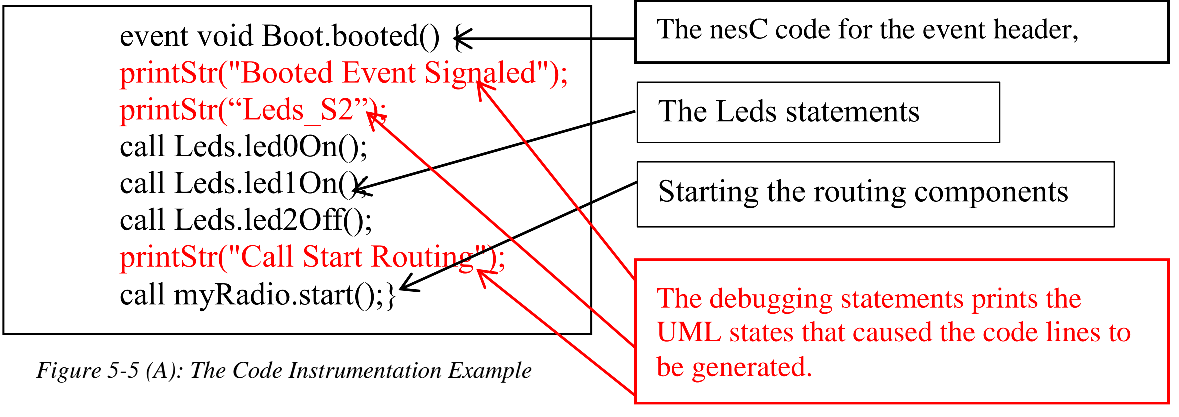


Figure 5-5 (A): The Code Instrumentation Example

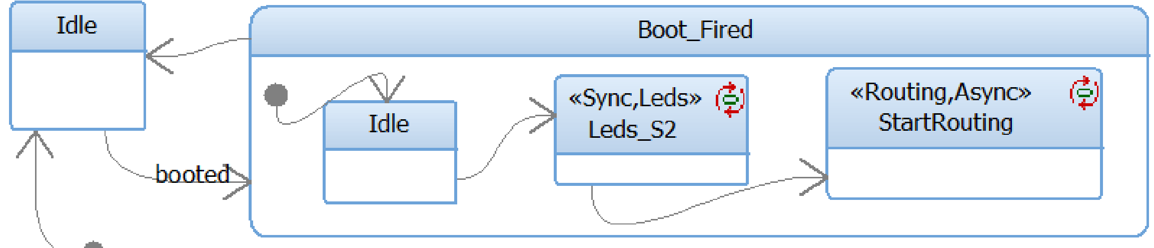


Figure 5-6 (B): The Model States

5.4.2 Instrumentation to Facilitate the Integration of the Simulator to the UML model

The integration of the Avrora analysis of the power consumption is achieved by using the debugging statements. The debugging statements are added to the code to indicate the following:

The code generator instruments the produced code with the debugging statements that are inserted in pairs, one to indicate the start time/cycle of a specific operation and the second one to indicate the end time/cycle of the operation. The debugging statements can be categorized as the following:

- ***Event Handlers Statements:*** One debugging statement is inserted at the beginning of the event handler and another statement is inserted at the end of event handler to displays the start cycle and the end cycle of the event handler respectively. The event handlers are the basic unit of the WSN software since the operating systems are event based.
- ***Spilt Phase Statements:*** One debugging statement is inserted at code position where the software calls the operation to display start cycle. The other debugging statement is inserted at the beginning of the event handler that is signalled when the hardware completes the operation, in order to display the time/cycle when the operation is completed.

The string that is printed by the debugging statements has two parts:

- ***Statement ID***: The statement ID refers to the event handler name, which exist in the UML model. The code generator is aware of the event handlers in the model and the equivalent nesC code event handlers. The code generator adds the event handler UML name so that the parsing tool calculates the power consumption that is caused by the event handler code and refers the results to the event handlers in the model. In the same manner, the debugging statement is inserted for the split phase operations. The statement ID will have the state name that represents the operation calling in state-chart diagram.
- ***Statement Type***: The type indicates whether the debugging statement is the start point or the end point of the either the event handler or a split phase operation. An example for integration debugging statements that are used to integrate the simulator with the UML is shown in Figure 5-7. Two debugging statement are inserted to indicate the start cycle and the end cycle of each operation. The debugging statement prints the statements once the code execution reaches the debugging statement location in the code flow. The simulator prints the debugging statement string, the node ID, and the time/cycle of the execution time and stores the output in a trace file (Figure 5-8). The trace file is used by the parsing tool to calculate the power consumption and generate the power consumption report.

The debugging statement used to indicate the starting and the ending of the event handlers

The debugging statement used to indicate the starting and the ending of the asynchronous process send

```
event void Timer0.fired() {
printStr("|Timer0Handler| St");
counter++;
printStr("|CallSend| St");
call AMSend.send(counter(;
printStr("|Timer0Handler| En");   }

event void AMSend.sendDone(message_t*msg, error_t
error){
printStr("|CallSend| En");
printStr("|SendDoneHandler| St");
 printStr("|CallTimer0| St");
 call Timer0.startOneShot(1000);
 printStr("|SendDoneHandler| En");  }
```

Figure 5-7: Example of Debugging Statement for Simulator Integration

```
Loading Rx.elf...[OK: 0.095 seconds]
Loading Tx.elf...[OK: 0.020 seconds]
=={ Simulation events }================
Node        Time   Event
----------------------------------------------------------
  0     8010824  |Booted| St
  0     8011224  |Booted| En
  0     8025530  |StartDoneHandler| St
  0     8025756  |StartDoneHandler| En
  1     8021638  |BootDone| St
  1     8022052  |BootDone| En
  1     8036330  |StartDone| St
  1     8036556  |StartDone| En
  1    15200241  |Timers0Handler| St
  1    15200360  |CallSend| St
  1    15202731  |CallSend| En
  1    15202835  |Timer0Handler| En
  1    15250673  |SendDoneHandler| St
  1    15250794  |SendDoneHandler| En
  ……………
```

Figure 5-8: The Trace File Output

5.5 Code Generation and Instrumentation Example

This section shows an example of the generated code for a UML model of a counter application. The application contains 3 nodes that have the exact software design. Each node is

activated every 5 minutes, adds 1 value to a local integer counter, and then sends the value to the neighbour node that displays the counter node through the LEDs. The model and the generated code (i.e. the configuration module and application module) are shown in Figure 5-9. The code is generated based on the following steps:

a. The model has four events;
   - ***Event booted***: The event is signaled once the node hardware completes booting
   - ***Event StartRoutingDone***: The event is signaled once the routing component is started
   - ***Event SendDone***: The event is signaled once the node completes sending the data
   - ***Event StartOneShot***: The event is signaled once the timer completes one cycle
b. The code generator parses the XML representation of UML for each event and creates a list of the states that are located in the UML event handler along with each state stereotypes.
c. For each state the code generator loads from the database the TinyOS component, the upper component, the interface, and the headers from the *config* table. Then, the loaded information is saved in the events class. The code generator uses the state stereotype as an indicator to the TinyOS component. For example, the stereotype "Sensor" in model will indicate the MDA3000CA TinyOS component. Table 4-1 in Chapter 4 shows all the model stereotypes and the equivalent TinyOS components.
d. The code generator creates the instances of each component and wires the component to operating OS support components by using the keywords *new, as,* and *components* and write the wired information is written in the configuration module.

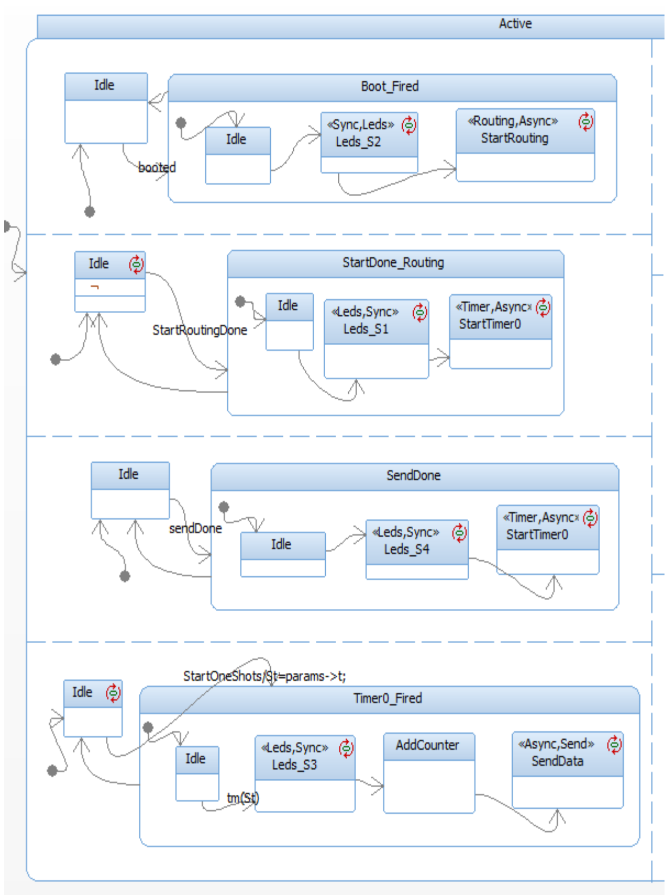
e. The components interfaces are written in the application module using the keywords *use*, *interface*, and *as*.

f. The code generator checks the *events table* in the database and loads the TinyOS events syntax, which are equivalent to the model events. Also, the TinyOS components that are associated with the events are loaded from the *config* table.

g. The event handlers' headers are written in the application module by using the loaded information from the previous step.

h. For each state, the code generator loads the nesC code syntax from the database that is equivalent to the state loaded from the table code module.

i. Before generating the final code, the code generator adds the debugging statement to generated code for validation procedure purposes and for the integration of the model and the simulator as explained in Section 5.4.)



Configuration Module

```
configuration  CounterAppC { }
implementation  {
components MainC, LedsC, Counter;
components ActiveMessageC as OSRadio;
components new TimerMilliC() as OSTimer;
components CollectionC as OSRouting;
components new CollectionSenderC(9) as OSSend;

Counter -> MainC.Boot;
Counter.Leds->LedsC;
Counter.myTimer -> OSTimer;
Counter.myRouting -> OSRouting;
Counter.mySend -> OSSend;
Counter.myReceive -> SRouting.Receive[9]; }
```

Application Module

```
Module Counter {
uses interface Boot;
uses interface Leds as Leds;
uses interface SplitControl as myRadio;
uses interface Timer<TMilli> as myTimer;
uses interface StdControl as myRouting;
uses interface Send as mySend;
uses interface Receive as myReceive;}
implementation  {
message_t sendbuf;
event void Boot.booted() {
printStr("Booted Event Handlers |Start|");
printStr("Booted Event Signaled");
printStr("Leds_S2");
call Leds.led0On();
call Leds.led1On();
call Leds.led2Off();
printStr("Call Start Routing");
("|PrintStr("Call StartRouting|Start
call myRadio.start();
printStr("Booted Event Handlers|End|");
}
event void myRadio.startDone(error_t error) {
printStr ("Booted Event Handlers |Start|");
printStr("|StartDone Event Signalled);
printStr("Leds_S1");
call Leds.led0On();
call Leds.led0On();
call Leds.led2Off();
PrintStr ("Start Timer0 |Start|");
printStr("StartTimer0");
call myTimer.startOneShot(1000);
printStr("Booted Event Handlers |End|");}
event void mySend.sendDone(message_t*msg,
 error_t {
printStr("Send Done Event Handler |Start|");
printStr("Send Data |Start|");
printStr("Leds_S4");
call Leds.led0On();
call Leds.led1Off();
call Leds.led2On();
printStr("Start Timer0");
("|printStr("StartTimer0 |Start
call myTimer.startOneShot(1000);
 printStr("Send Done Event Handler |End|");
}}
event void myTimer.fired() {
printStr("Timer0 Fired Event Handlers |Start|");
("|printStr("StartTimer|End
printStr("Leds_S3");
call Leds.led0Off();
call Leds.led1Off();
call Leds.led2On();}
printStr("Send Data");
printStr("Send Data |Start|");
call mySend.send(&sendbuf, 56);{
event message_t*
myReceive.receive(message_t* msg, void* payload,
uint8_t len) {};
```

Figure 5-9: Code Generation Example
Debugging statements (red)
Keywords (blue)

5.6 Validating the generated Code

The aim of the code validation procedure is to confirm that the generated code has the same semantics as the UML models. The validation procedure occurs by comparing the design semantics in both layers (i.e. the code layer and the model layer) and tracing the signals between the components in both layers.

In order to demonstrate the code validation procedure, the generated code is instrumented with the debugging statements together with the call trace feature in Avrora simulator. The debugging statements in the code display the state names that generated the code lines, as explained in the previous section. The call trace option displays the signals between the OS support components and the application component and records the signals in a log file. The debugging statements and the trace of the signals are saved in a log file. For the validation procedure, the log file is compared to the executed sequence that shows the current state of the software and the signals between the components.

In order to demonstrate the code validation procedure, the validation procedure for a simple counter application is shown in Figure 5-11. The counter application contains 2 nodes: the sender node and the receiver node. The sender node is activated every 1 minute and then adds value 1 to a counter and sends the value to the receiver node. The execution sequence for the sender node was generated through UML model execution of the design (see Figure 5-11). The sequence diagram is for 1 cycle. The sequence diagram is split into 4 sections due to the size limitations. The sequence diagram shows that the sender node is booted followed by triggering the booted event. Once the booted event is triggered, the software turns the *Leds_S2* on by sending the signal to the LEDs support component, and then the software starts the routing

component by sending the proper signal to the routing component. Once the routing component is started, the routing *startdone* event is signaled.

In the *startrouting* event handler, the software turns on another group of LEDs and then initiates a timer. Once the timer expires, the timer signals the *timer* event. In the timer handlers, the software turns on other groups of LEDs, add value 1 to the counter, and sends the data packet. Once the sending process is completed, an event is signalled to indicate the end of the sending process. In the event handler, the software turns a group of LEDs on and starts the timer again for the next cycle, as it appears in the executed sequence diagram. At the send node, the LEDs can be used to display the status of the code where each even handler has a specific LEDs turned on. The simulation results that are recorded in the log file is shown in Figure 5-12. The log file contains the output debugging statements from Avrora, which displays the UML states ID from the generated code. In addition, the log file displays the call the application components and OS support components. The comparison shows that the instrumented code semantics matches the sequence diagram behaviour.

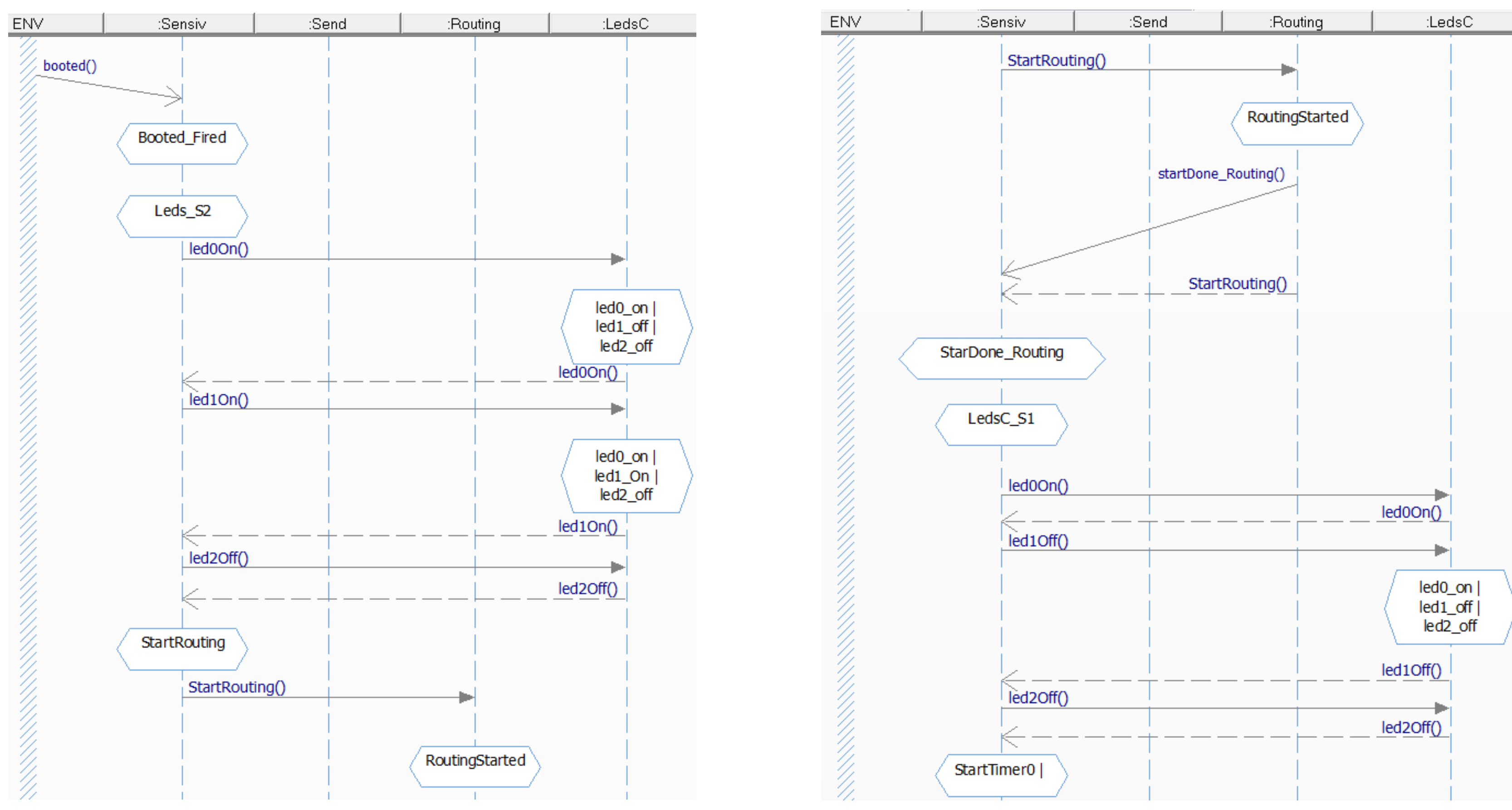
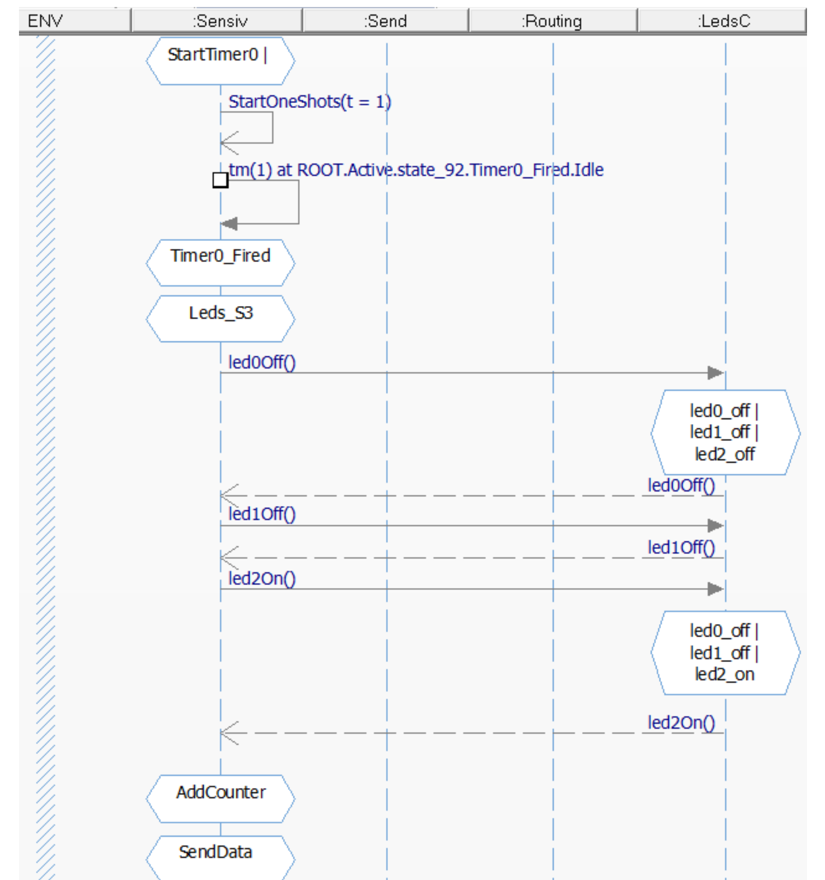
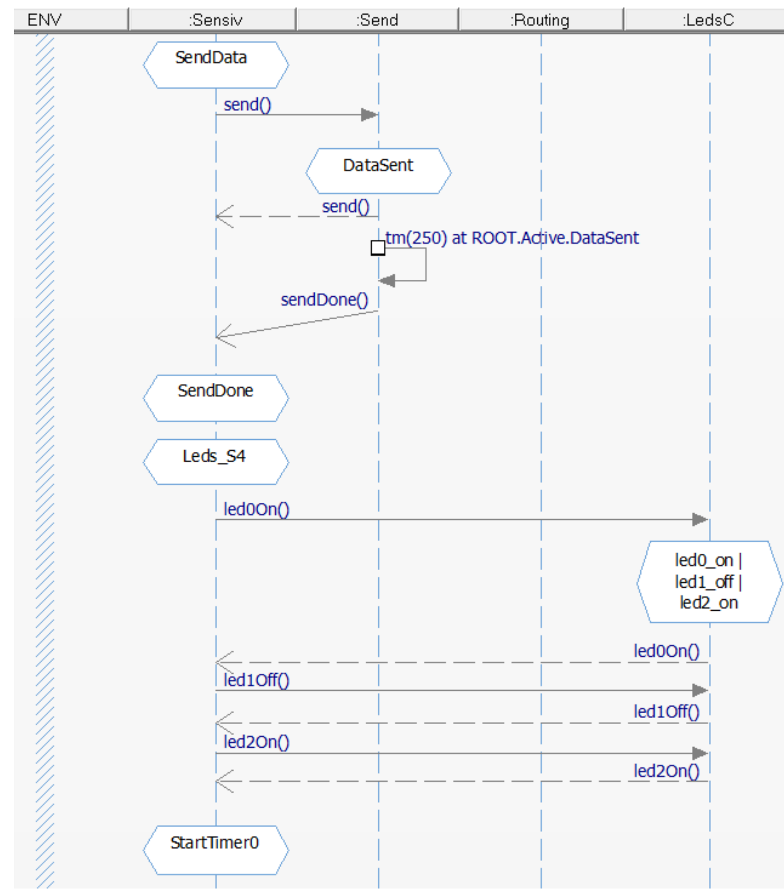


Figure 5-10: Executed Sequence Diagram for Counter Sender Node (Part -A)





…….

Figure 5-11: Executed Sequence Diagram for Counter Sender Node (Part B)

```
Loading RX.elf...[OK: 0.112 seconds]
Loading TX.elf...[OK: 0.026 seconds]
=={ Simulation events }=========================================================
Node        Time   Event
----------------------------------------------------------------------------
   1      8012253  |Booted Event Signaled|
   1      8012327  |Leds_S2|
   1      8012342  on  on  off
   1      8012505  |Call Start Routing|
   1      8012507      --> CC2420CsmaP__SplitControl__start
   1      8012805      <-- CC2420CsmaP__SplitControl__start
   1      8017193      --> #13 0x0030
   1      8017290          --> CC2420SpiP__Resource__request
   1      8017486          <-- CC2420SpiP__Resource__request
   1      8017524      <-- #13 0x0030
   1      8027271  |StartDone Routing Event Signalled|
   1      8027345  |Leds_S1|
   1      8027360  on  off off
   1      8027474  |Start Timer0|
   1      8027517          --> VirtualizeTimerC__0__startTimer
   1      8027560              --> SchedulerBasicP__TaskBasic__postTask
   1      8027604              <-- SchedulerBasicP__TaskBasic__postTask
   1      8027614          <-- VirtualizeTimerC__0__startTimer
   1     15190792  |Timer0_Fired|
   1     15190866  |Leds_S3|
   1     15190881  off off on
   1     15190974  |Add Counter|
   1     15191088  |Send Data|
   1     15191123              --> CC2420ActiveMessageP__AMSend__send
   1     15193363              <-- CC2420ActiveMessageP__AMSend__send
   1     15193367              --> SchedulerBasicP__TaskBasic__postTask
   1     15195475              <-- SchedulerBasicP__TaskBasic__postTask
   1     15195495          <-- VirtualizeTimerC__0__fireTimers
   1     15195521      <-- SchedulerBasicP__TaskBasic__runTask
   1     15195555      --> SchedulerBasicP__TaskBasic__runTask
   1     15195593          --> Atm128AlarmAsyncP__0__Counter__get
   1     15195625          <-- Atm128AlarmAsyncP__0__Counter__get
   1     15195680      <-- SchedulerBasicP__TaskBasic__runTask
   1     15240914      --> SchedulerBasicP__TaskBasic__runTask
   1     15241032          --> AMQueueImplP__0__AMSend__sendDone
   1     15241048              --> AMQueueImplP__0__sendDone
   1     15241092                  --> BlinkToRadioC__AMSend__sendDone
   1     15241302  |SendDone Event Signalled|
   1     15241418  |Timer0_Fired|
   1     15241492  |Leds_S4|
   1     15241512  on off on
   1     15241626   Start Timer0
```

Figure 5-12: The screen shot Node1 log file

5.7 Summary

This chapter introduced the structure of the code generator, which generated the required code for the simulation and generated the necessary configuration for the simulator. The code

generator generated the code by parsing the XML representation of the UML model and loaded the required nesC syntax from a central database. The model provided the code generator with the information about the event handlers, events, and the design components while the database provided the code generator with the TinyOS components, parameters, methods, header files, and the events syntax. The code generator prompted the designer for additional information about the design, such as the topology, the platform, and the battery capacity since this information varied from one design to another and do not existed in the database. The code generator instrumented the generated code in order to verify the code and integrate the simulator results to the model. The code verification occurred by comparing the instrumentation output and the tracing of the components calls to the executed sequence diagram.

6 Integration of the Simulator and the UML Models

6.1 Introduction

This chapter presents the integration between the Avrora network simulator and the UML software models so that the software WSN software designer can modify the UML models to improve the WSN performance. In the previous chapters, this thesis demonstrated how to represent the design and the analysis of the power consumption at the modeling layer such that the designer is able to verify the design and calculate the power consumption at the early stages of the design process. However, the presented approach in the previous chapters, is not capable of calculating the power consumption of the communication process, which consumes 60% of the total power consumption. Therefore, there is a necessity to calculate the power consumption of the communication process and integrate the results with the UML model for the design so that the designer has a complete estimation of the power consumption at the modeling layer. Based on the power consumption that was developed at the UML model (presented in Chapter 4) and the simulation power consumption analysis (Chapter 5) , the designer should be able to improve and correct the design at the modeling layer.

The code generator produces the nesC code, network topology, and the simulator configuration, required by the network simulator (as discussed in Chapter 5). Also, the code generator instruments the generated code in order to help facilitate the integration of the simulation results back to the model (this will be discussed in more detail in section 6.5.) A parsing tool was developed to parse the power log file for each node, the debugging statements trace file, and generate a power consumption report. The generated power report enables the designer to integrate the simulator results back to the UML model. This process is captured in Figure 6 1.

The rest of the chapter is organized as following: Section 6.2 describes the TinyOS operational nature to illustrate the structure of the code instrumentation. Section 6.3 describes the communication process between the WSN nodes and its non-deterministic nature, which justifies the necessity of using a WSN simulator for the power consumption of communication process. Section 6.4 shows the instrumented code, which eases the integration. Section 6.5 explains the parsing tool that enables the designer to integrate the simulation analysis results back to the model. Section 6.6 contains an example of the process for a simple TinyOS application, and Section 6.8 contains the summary.

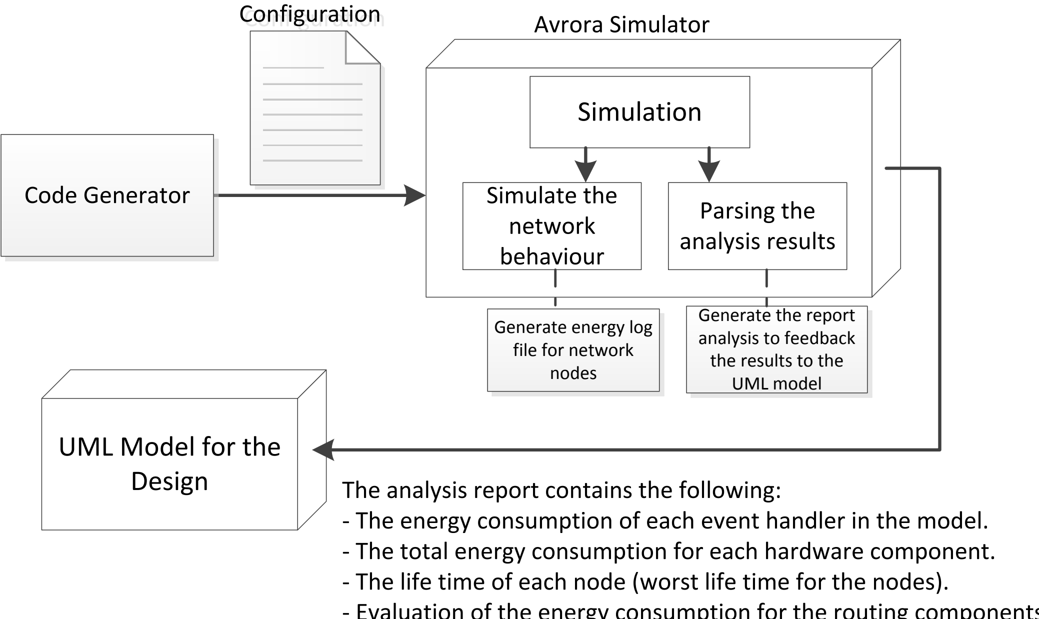


Figure 6-1: Integration of the Simulation and the UML Models

6.2 The Parsing Tool

As explained in the Section 5.4.2, the debugging statements tool *Avroraprint.h* prints the debugging statements that are inserted by the code generator to indicate the start CPU and the end CPU cycle of each event handler and each operation. The printed results contain the node ID where the debugging statement was printed, the operation/event name as indicate in the UML

layer (i.e. UML name), the CPU cycle of the debugging statements, and the statement type (i.e. start or end). The printed statements are stored in a trace file.

The power consumption tool of the Avrora simulator generates a power log files for each single node in the topology. The power log file contains the amount of power that is drained from the battery for the OS support components (CPU, Red LED, Green LED, Yellow LED, Radio, Sensor Board, and Flash). The log file is updated with a new entry to indicate the amount of power consumed when one of the hardware resources is activated. In addition, the simulator prints the CPU cycle of each entry in the file.

A parsing tool was developed to parse the power log files for each node and the debugging statement trace file. The parsing tool searches the trace file, extracts all the operation names of each node and the start/end cycle of each operation, and then stores the information in an array. For each log file, the parsing tool finds the entries that fall in the start and cycle range for each operation/event and total the power consumption for each hardware resource. For example, node 1 debugging statements indicate the *CallSend* operation started at the CPU cycle 22406624 and ended at 22449918. The debugging statements keyword *St* and *En* indicate the start and the end of the operation respectively. The parsing tool searches for log file of node 1 for all entries that lay between the range 22406624 and 22449918. The parsing tool creates an integer variable for each hardware resource (CPU, RED, Yellow, Green, Radio, Board, and Flash) and stores the summation of the energy consumption of each component within the cycle range. The total energy consumption for each component will the summation of all numbers displayed in the resource column (see Figure 6-2).

The parsing tool generates a report that contains the following information:

- The power consumption of each operation and each event handler. The parsing tool refers to the operation and event handlers by using the syntax that is used in the UML model.
- The rate of calling each operation.
- The total power consumption of each hardware resource.
- The life time of each node (provided by the simulator) so that the designer can conclude the worst life-time among all nodes.

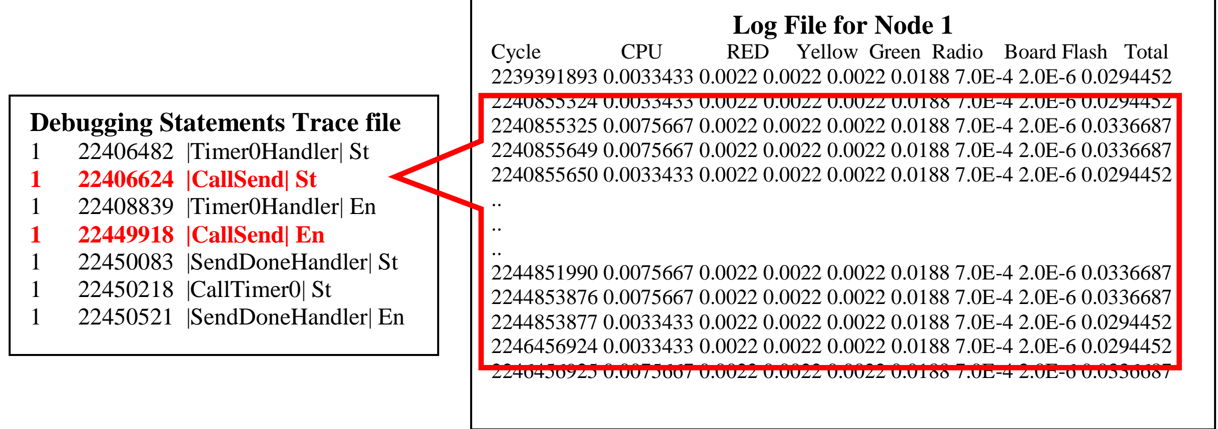


Figure 6-2: Parsing Log files and Trace file

The receiving operation is signalled by packet once the packet reaches the node. Once the receiving operation is completed, the *myradio.receive()* event is signalled. Therefore, the receiving operation occurs independent of the destination node code, although the code has signalled an event to indicate the completion of the receiving. Therefore, there are no entries in the log file to indicate the starting of the receiving operation since the operation is initiated by the data packet. However, there are entries to indicate the completion of the operation (i.e. the received event handler). In order to calculate the power consumption of the receiving operation, the parsing tool

traces the number of the receiving event handlers and multiplies the value by the power consumed by the receiving operation that is published in the Avrora documentation.

6.3 Integrating Simulation results with UML Model Example

This section presents the analysis of a counter application, which contains 3 nodes. The sender node is activated every 5 minutes, adds 1 value to a local integer counter, and then sends the value to a receiver node. Figure 6-3 presents a portion of the sender node model. The model contains two event handlers: *Timer0_Fired* and *SendDone*. In the *Timer0_Fired* event handler, the code adds value 1 to a counter and initiates the send operation. Once the send operation is completed, *sendDone* event is signalled and the event handler *Timer0* is started again.

The code generator inserts four pairs of debugging statements, two pairs to indicate the cycle boundaries of the two event handlers, one pair to indicate the cycle boundaries of the send operation, and one pair to indicate the cycle boundaries of the timer. The topology configuration contains three nodes where each node is a sender and a receiver as well. The same code is applied for all nodes in the network. Avrora simulates the code and prints one trace file, which contains the debugging statements for all operations of all nodes and power log file for each node. The parsing tool analyzes the files, as previously explained, generates the report that enables the designer to integrate the results with the UML model.

In some WSN applications, some nodes have different code designs than others. In this case, a separate UML design is created for each node and multiple copies of the code is generated which correspond with the deployment of the 3 nodes. The procedure of the parsing tool does not change since parsing tool analyzes the tracing file and the power log file.

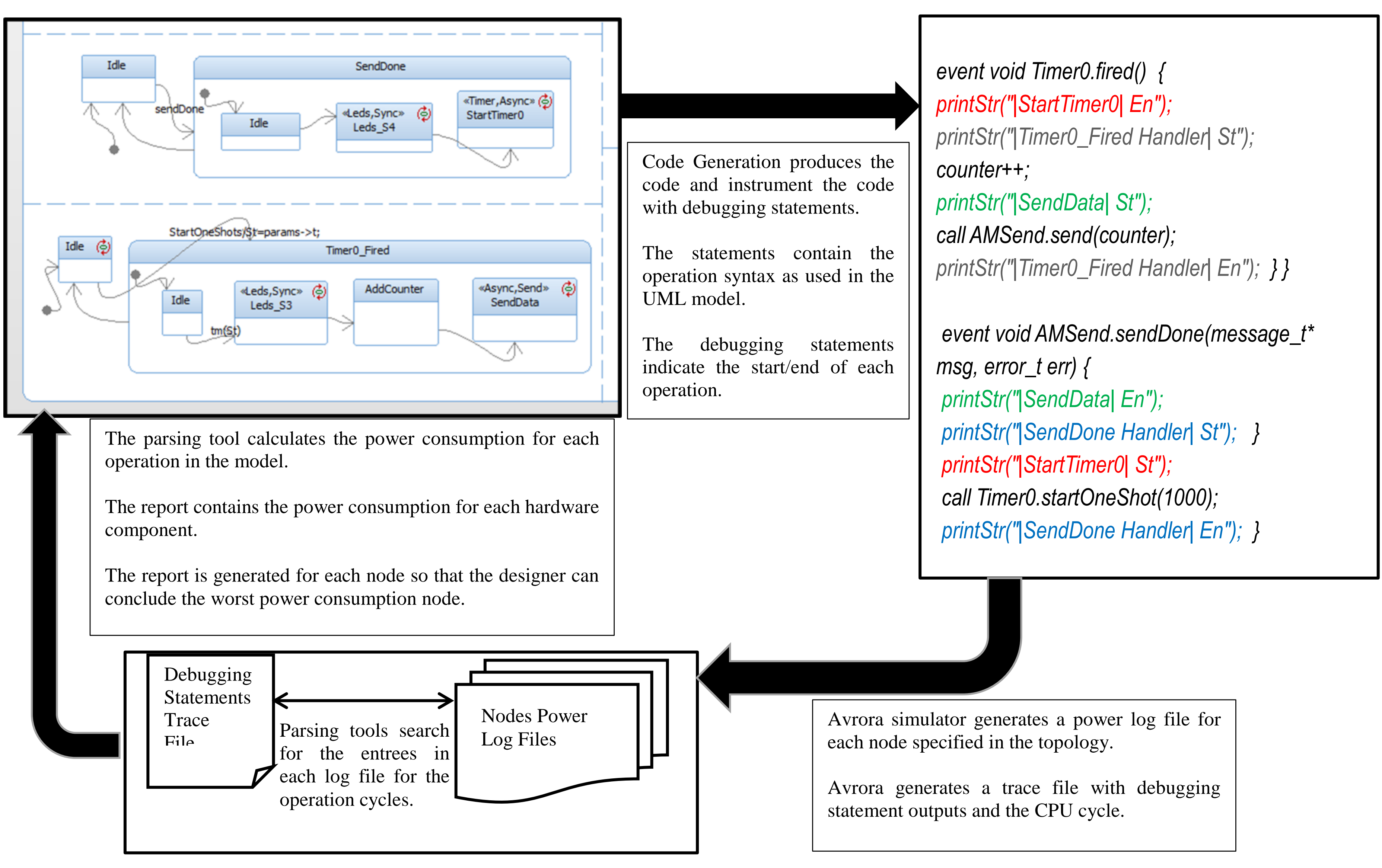


Figure 6-3: Power Consumption Analysis of Counter Application

6.4 Summary

This chapter explained the integration between WSN simulators (i.e. Avrora simulator) and UML models that capture the design structure and the design behaviour. The UML model is able to analyze the power consumption of the deterministic components, such as LEDs, ADC, and sensors. However, the UML model is not capable of analyzing the power consumption of the communication process due to the non-deterministic nature of the multi-hop communication process. Avrora simulator provides the power consumption tool that produces the power consumption log file for each node in the network. The log file contains the power consumption of each hardware resource. The log file is updated with a new entry at each executed CPU cycle. In addition, Avrora simulator supports a debugging statement tool that prints out the string of the debugging statement when the complier reaches the statement location in the code.

To achieve the integration between the Avrora simulator and the UML model so that the designer is able to predict the power consumption at the UML model layer, the generated code was instrumented with the debugging statements to indicate the start and the end cycle of each operation. The debugging statements contain the operation syntax, as shown in the UML model. In addition, a parsing tool was implemented to analyze the power log of all nodes and calculate the power consumption of all operations and event handlers, and then generates a report. The generated report contains the power consumption for all nodes and the breakdown of the power consumption for each operation in the node so that the designer can evaluate the worst node in the terms of power consumption and integrate those results in the UML model. The report contains the total power consumption for each hardware resource so that the designer can evaluate the performance of the hardware resource.

7 Evaluation

7.1 Introduction

This chapter evaluates this thesis approach by discussing the design process for a typical collector WSN system called SensIV. For the SensIV system design, the chapter explains the system architecture, the system requirements, the design process, and the power consumption analysis using this thesis approach.

As presented in the previous chapters, the system design of a wireless sensor system is captured using UML object and state diagrams guided by a set of UML design patterns. If the wireless sensor system consists of single hop communications the power consumption model is deterministic and can be easily specified at the UML modeling layer. In these scenarios the executed UML diagram features were leveraged to validate the design against the system requirements and analyze the power consumption of the hardware components. For multi-hop sensor networks the communication process of the WSN system has a non-deterministic behavior due to the stochastic nature of the routing protocol algorithms. In the non-deterministic scenario, it is more suitable to analyze the power consumption of the sensor network using a network simulator. In order to use the network simulator a code generator is required which produces the code for the simulator as well as the network configuration. Also, a parsing tool was implemented to analyze the power consumption results from the simulator and help the software designer to integrate these results back to the UML model so that the designer can improve the software design of the WSN .

For the non-deterministic systems, the power consumption analysis is developed by using the UML modeling power consumption analysis approach in combination with the power consumption analysis of the simulator. For the deterministic systems, the power

consumption analysis at the modeling level is sufficient to estimate the power consumption of the system without the necessity of using the simulation. However, the designer can use the code generation tool to produce the code for the deterministic system and validate the produced code as part of software development cycle.

The rest of the chapter is organized as follows: Section 7.2 explains the alternative that designer can utilize to improve the power consumption. Section 7.3 focuses on the power analysis of the SensIV system which consists of power consumption analysis at the modeling layer, power consumption using the Avrora simulation tool, and the integration of the power analysis simulator results to the SensIV UML model.

7.2 Power Consumption Improvement

This chapter shows how the designer can use the approach developed in this thesis to evaluate the enhancements that can be introduced to the design to improve the power consumptions. The enhancements can be categorized into 2 main categories:

Modifying the software logic: The modification of software logic takes place at the modeling layer entirely since the logic is captured by using the UML modeling patterns that was introduced in Chapter 4. Meantime, the software logic has a strong impact on the power consumption of the sensor node components and the power consumption of the network. The impact of software modification on the sensor node power consumption is analyzed by the executed state-chart methodology that was presented in Chapter 4 while the impact of the network power consumption was analyzed by using the integration between Avrora simulator and the UML model that was presented in Chapter 5 and 6. This chapter presents four types of the software logic modification that can be used to improve the power consumption of the sensor node components and the power consumption of the network. An example of the power

consumption of a sensor node is demonstrated by analyzing the power consumption for two scenarios of software logic that controls the ADC unit. The analysis for the two scenarios was developed at the modeling layer using the executed state-chart feature as explained in section 7.3.3. The software logic that impacted the network power consumption is demonstrated by two scenarios for the data collection protocol and by modifying the event handler structure as explained in section 7.3.4. The data collection protocol is implemented by the software logic that was captured by the UML design patterns. However, the protocol controls the amount of data packets sent by each node and amount of data packets that is forwarded across the network and ultimately the power consumption of the network. The event handler structure contains the calls for the components to complete a specific task. Based on the network operation, some of the events handlers can be signaled often compared to other event handlers. The network analysis calculates how often each event was signaled while the network was operating. Therefore, the designer can modify the structure of the event handler to reduce the hardware resource calls in this particular event handler.

Modifying the Network Configuration: The network configuration influences the power consumption of the network. For instance, the Low Power Listening (LPL) mode controls radio communication between the nodes and is initiated when the node is booted. There are no major changes in the UML model in order to initiate the LPL, however, it has to be presented in the UML model so that the code generator can produce the required code lines for the LPL as explained in Section 7.3.4.4.

As explained in the Chapter 6 that the parsing tool analyzes the simulator results and generates a power consumption report that enables the designer to feedback the simulation

results back to the model (see Chapter 6). The feedback loop enables the designer to evaluate the modification and use alternatives to improve the design for power consumption.

7.3 SensIV System Design

7.3.1 SensIV System Requirements and Specifications

The aim of the SensIV system is to monitor the temperature of a vineyard field. SensIV is considered as a typical collector WSN system consisting of a set of sensor nodes that collect temperature data across a field and transmit this data to a collection point using wireless signals. The nodes communicate with each other wirelessly until the data reaches the gateway where the temperature values are stored in the database for further analysis.

SensIV system consists of 10 nodes, each node contain the following:

- ***Four temperature sensors:*** The sensors used are thermal sensors of type LM135 [4]. The sensor's accuracy is +/- 1°C and can measure a temperature range in between -55°C and 150°C.
- ***MDA300CA acquisition board***: The MDA300CA data acquisition board from Crossbow has expansion connectors for 7 single-ended and 4 differential ADC channels.
- ***Wireless Iris node:*** The Iris wireless node uses the Atmel processor with 128 KB program memory and 8KB RAM. In terms of radio communication, it uses 2.4GHz (IEEE802.15.4) with a range of 500 meters. This type of node also supports a low-power mode of operation for the micro-processor and radio.
- ***Solar cell:*** A solar cell is used to recharge the 2 AA alkaline batteries that power the sensor node. The system was designed for continuous operation with sunlight conditions encountered in Southern Ontario.

- ***CTP Routing Component:*** CTP is an address free protocol that maintains a tree routing topology among the sensor network to the collector. The version of CTP used is the one available in the TinyOS deployment.

The system requirements are as following:

- ***Sensing the temperature:***
  - The sensor is activated every 5 minutes.
  - The system turns the ADC on 1 minute before sensing.
  - The system signals the 4 sensors.
  - The system turns the ADC off.
  - The nodes send the data packages to the neighbour node.
- ***Maximum overnight power consumption (7 hours) is 2000 mAH:*** 2000 mAH is the maximum capacity of the battery until the solar cell charges the battery during the day.

The sensing portion of the design has a deterministic behaviour since the sensor is activated every 5 minutes and the power consumption characteristics of the sensor and ADC can be mathematically modelled. The sensing activity includes turning on the ADC for one minute (this is known as the sensor warm-up period), followed by signalling the four sensors to commence the sensing, and finally turning off the ADC. The communication process among the nodes has a non-deterministic nature since SensIV is a WSN and the CTP routing protocol is continuously trying to maintain a tree topology in a stochastic noisy environment.

7.3.2 UML Models for SensIV System

The service component pattern that was introduced in Section 4.3.1 captures the system components by using defined stereotype classes. The UML model for SensIV contains

four stereotype UML classes: the radio component; the routing component, the acquisition board component, and the LEDs component as well as the SensIV software class. The composite class diagram has the four class diagrams that capture the behaviour of OS support components (*itsRouting, ItsSensIV, itsMDA,* and *itsLEDs*). Also, the composite class contains *itsSensIV* class which contains the design software component that controls the system behaviour (see

Figure 7-1). The five classes capture one sensor node design of WSN system.

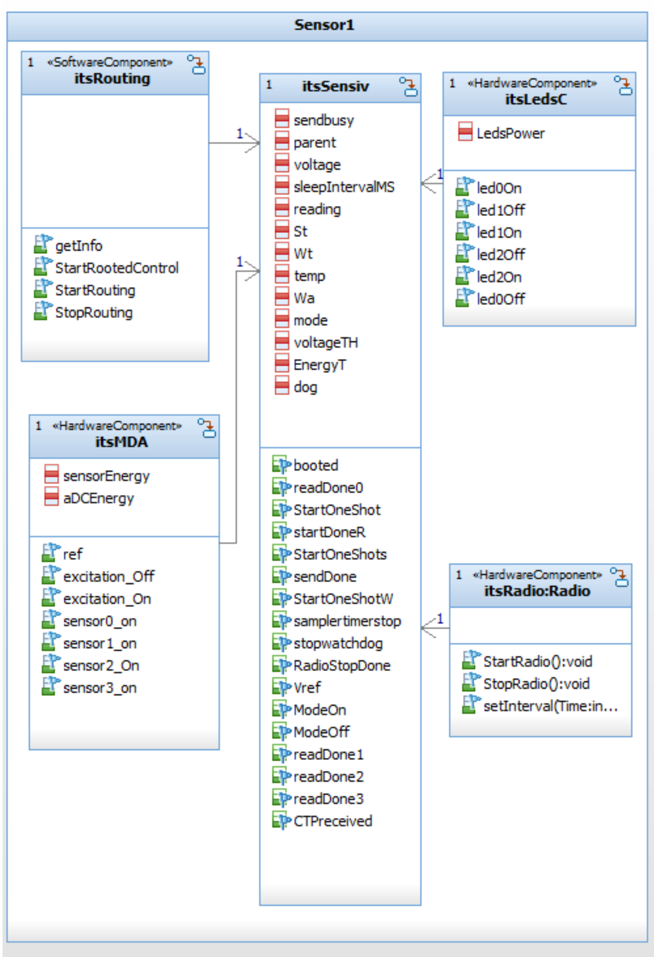


Figure 7-1: Service Components UML Model for SensIV

As mentioned in Section 4.3.1 that each stereotype class contains the methods and parameters that are provided by the components and are essential to manage the OS support components. Figure 7-2 shows an example of the routing stereotype class. The classes contain three methods:

- ***startrouting***: The system software signals this method to enable the routing protocol to build the routing tree.
- ***stoprouting:*** The system software signals this method to stop the routing protocol. In most cases this method is used to restart the routing protocol.
- ***getgnfo:*** The system software uses this method to collect information about the parent node.

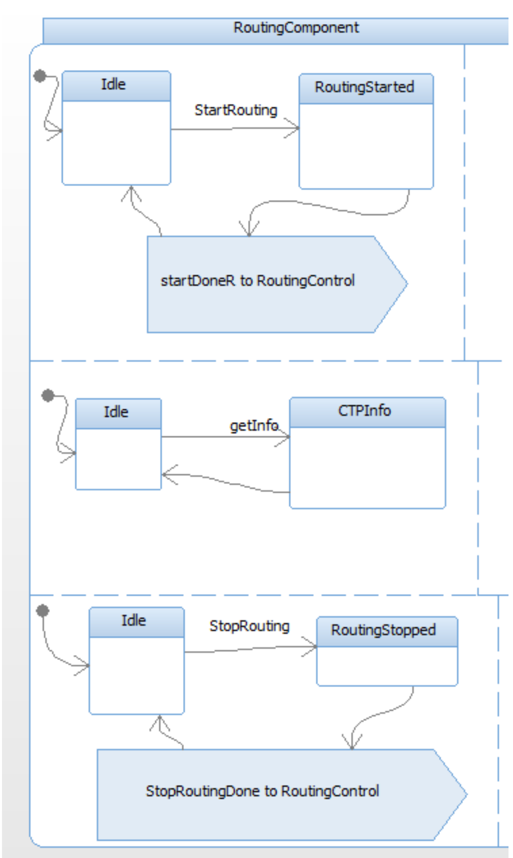


Figure 7-2: Routing OS Component State-Chart

The methods *startrouting, stoprouting,* and *getinfo* are signaled by the SensIV software component based on the software workflow. Once the state-chart component completes the operation, the service component signals an event back to the application software.

The software behavior of SensIV is captured by *itsSensIV* class using the event handler pattern that was introduced in Section 4.3.2. The event handler pattern captures the events and the event handler structure. SensIV classes capture all the events that are signaled by the stereotypes classes and the event handler structure that contains node behavior once the event is signaled. The events that are captured by SensIV are signaled by the followings components:

- ***Timers:*** Timers generally control the flow of the entire system. Once the timer expires, the timer signaled the event *Timer_fired*. The event handler contains the execution steps that the node will flow. In SensIV, we have 2 main timers:
  - ***Timer1***: The timer is started once the node is booted and it controls the main cycle of the node. Every 5 minutes the time expires and signals the event *Timer1_fired*. In the event handler, the software wakes up each node and starts to perform the sensing operation.
  - ***Timer2:*** The timer is started when the software turns the ADC on in order to warm up the hardware. Once the timer expires, the event handlers signal the event to start sensing.
- ***Sensors:*** The sensor signals the software to indicate the completion of the sensing process. Due to the hardware limitation, the sensors have to be signaled sequentially. The sensors signal the events *readDone* that are signaled by the sensor classes (see Figure 7-3).

- ***Routing Protocol***: The routing protocol signals three events:
    - ***StartRotuing_Done***: To indicate that the routing component is started.
    - ***StopRouting_Done***: To indicate that the routing component is stopped.
    - ***GetInfo:*** To indicate that the reading of the parent node info is completed.

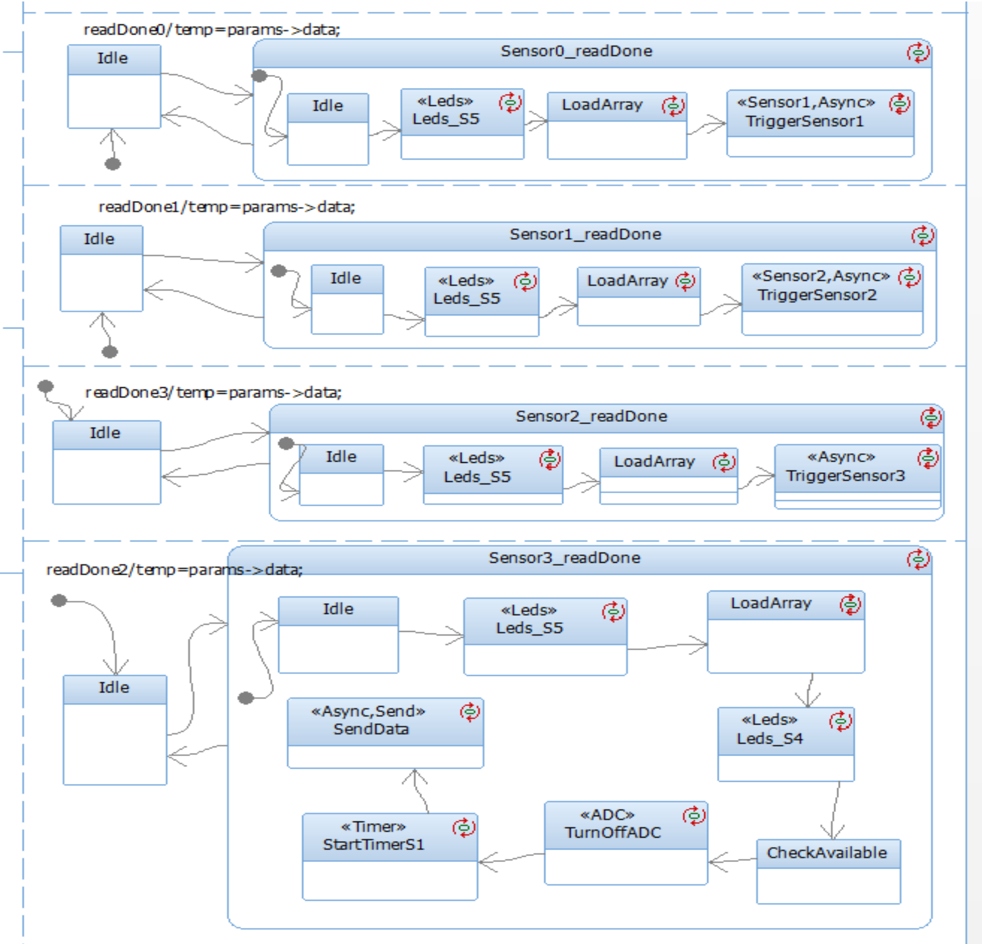


Figure 7-3: Portion of SensIV State-Chart

- ***Radio:*** The radio unit signals the software to indicate that the sending operation is completed.
- ***Booted:*** The hardware signals the software to indicate the completion of the booting process. Normally, the booted event is the first event to be signaled.

- The association between the classes is captured by the Association of the Application and OS support Component pattern introduced in Section 4.3.3. The association captures the component wiring for the design. The component association enable IBM rational rhapsody to animate the UML diagram to complete the analysis. The association is completed by creating the *ClassPart* that links the SensIV application with the rest of the stereotypes classes. The created class parts are shown in the panel diagram (SensIV Class/Parts) (see Figure 7-4).

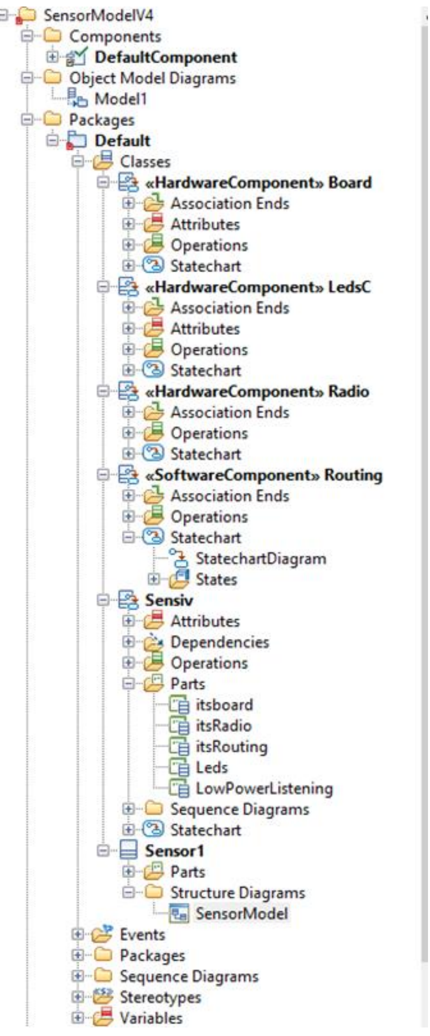


Figure 7-4: SensIV Panel Diagram

7.3.3 SensIV Analysis at the Modeling Layer

As introduced in the previous chapter, the UML model execution feature was leveraged to validate the design workflow requirements and calculate the power consumption of the sensor node. The analysis at the modeling layer relies on the instrumentation for the state-charts of the stereotype classes, as explained in Chapter 4.

7.3.3.1 System Design Validation Procedure

The system design is validated against the system requirements by using the comparing sequence diagram feature that is supported by IBM rational rhapsody. The designer creates the assumed sequence diagram that contains the expected behaviour of the system to fulfil the requirements. The assumed sequence diagram is compared to the executed sequence diagram. The comparison shows if both sequence diagrams have the same order of signals between the components, states, and timing. The complete sequence diagram association is shown in Appendix IV.

7.3.3.2 Sensor Node Analysis

Based on SensIV system design, the components ADC, sensor, and LEDs are activated on a regular basis. The power consumption analysis for those components can be developed at the modeling layer since signalling those components occur on a deterministic manner. As mentioned in Section 4.5.1, the state-charts are instrumented with the execution time and the power consumption constraints for the hardware resources (i.e. ADC, sensors, and LEDs). The time and the power annotations are calculated based on real-time measurements [35] and shown in Table 7-1.

Component	Current mA/sec
ADC excitation	13
Sensor	1
LED	2.2

Table 7-1: Power Consumption Constrains

In order to demonstrate the advantages of performing the power analysis at the software-modeling layer, three scenarios were presented for controlling the ADC and each have a significant impact on the power consumption of the sensor unit.

To recap, recall that in the SensIV application each node supported 4 sensors. In the first scenario (S1), the application signals the sensor excitation to turn on and then signals the ADC to read the sensors one after another. While the 4 sensors sense the temperature, the sensor excitation is kept running. Finally, after the 4 sensors finish sensing, the excitation is shut down. In the second scenario (S2), the application signals the sensor excitation and then reads the first sensor value. After the first sensor finishes sensing, the application shuts down the sensor excitation, saves the sensed data, and turns on the excitation to prepare the second sensor for reading. In the third scenario (S3), the same design flow happens as in the first scenario, except the designer forgets to shut down the sensor excitation after completing the sensing. Forgetting to shut down the sensor excitation is one of the common design mistakes in WSNs. Scenarios 1 and 2 offer design alternatives while scenario 3 is simply a design mistake.

The analysis was run for the three scenarios for a 7 hours period of lengths, which is the period when the batteries are not being charged with the solar cell. The node was activated every five minutes to trigger the sensors. The results of the analysis are shown in Figure 7-5. The X-axis represents the time component while the Y-axis shows the accumulative value of the consumed current. The analysis was run for the 3 scenarios for 7 hours and the node was activated every 8 minutes to trigger the sensors. The results showed that the first scenario had

consumed 3.16 mA or 0.05 mA/hr., and the second scenario had consumed 11.89 mA or 0.19 mA/hr. while the third scenario had consumed 215 mA or 3.4 mA/hr.

The designer has the choice to use the simulator to analyze the deterministic and the non-deterministic components or to analyze the deterministic component only (i.e. power consumption of communication process) since the power consumption of the deterministic components are analyzed using the UML model execution technique.

Using the simulator to analyze the power consumption of the communication process only will require the development of a sub-system that contains the communication control methods (i.e. sending and receiving) while the rest of the design is abstracted out. In this case, the total power consumption of the node will be the power consumption of the deterministic components (i.e. sensors, ADC, and LEDs) that are analyzed by the model in addition to the power consumption of the communication process that is analyzed by the simulator. The deterministic power consumption analysis for 7 hours is shown in Table 7-2.

7.3.4 Power Consumption Analysis of the WSN Network

SensIV design relies on CTP routing protocol to forward the sensed data to the one collector node where the information is further processed and displayed to the user. As explained in previous chapters, CTP protocol uses ETX method to establish the routing tree, which leads to the non-deterministic nature of the communication process. In addition, the data loss and the routing packages that exchanged between the nodes leads to the unpredictable power consumption of the communication process. Therefore, in order to complete the power consumption analysis, the network simulator Avrora is required.

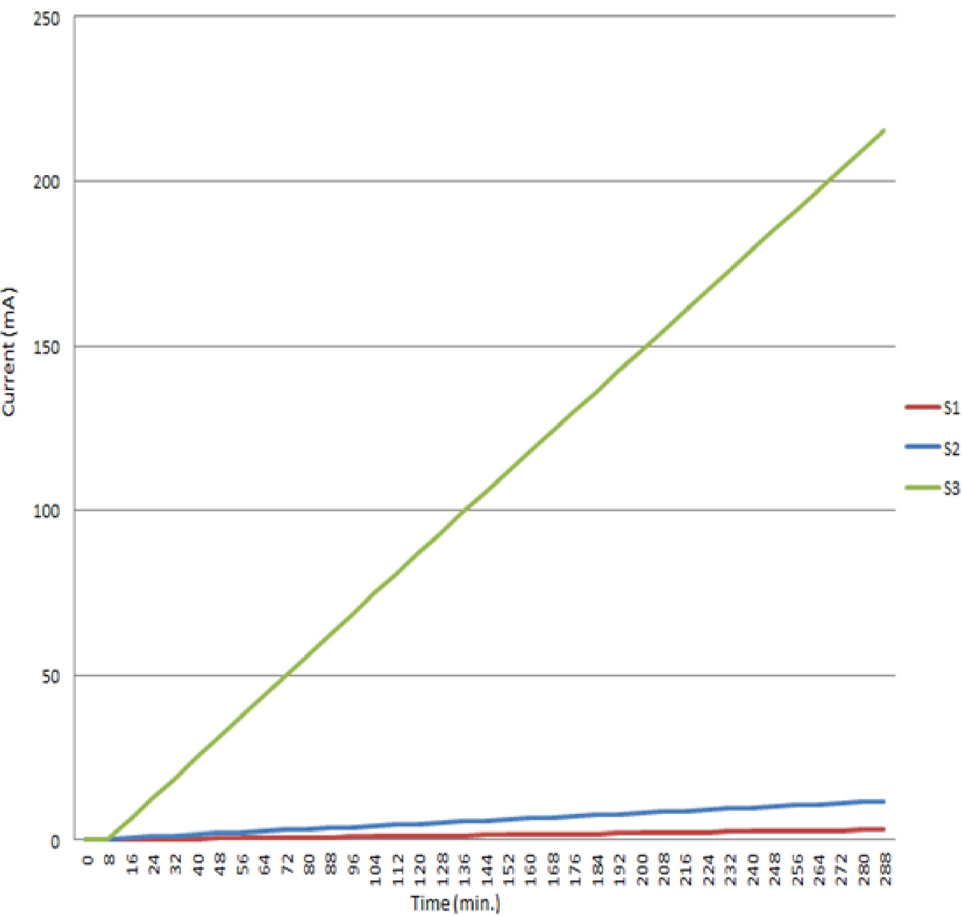


Figure 7-5: Power Analysis for Three Scenarios of Controlling the ADC

Component	Current mA	Current mAH	Joules
Sensors	11830 mA	31 mAH	334
ADC	1820 mA	0.5 mAH	5.4
LEDs	3900 mA	1.11	10.8
Total		32 mAH	350

Table 7-2: Power Consumption Analysis for SensIV Determintsic Components

IBM rational rhapsody exports the XML representation of the UML model. The code generator parses the XML representation and prompts the designer for the following design details that are required for the simulation:

- ***The routing protocol:*** The routing protocol that designer will use for system.

- ***The topology structure:*** The topology structure is defined by a text file, which contains the physical location of the nodes defined as X Y Z coordinates. The designer places sensors in the field to achieve the best monitoring resolution. The monitoring resolution is directly proportional to number of sensors in a square area. Additionally, the network connectivity is directly proportional to the number sensors in the square area because if the one of the nodes dies, the routing protocol can re-build the tree using another node within the child node area. This thesis does not focus on the impact of the topology structure on the power consumption. Therefore, the topology structure used was provided with SensIV system documents. The topology structure configuration for the simulation is shown in Figure 7-6. The physical sensor location is displayed on a 2-D google map in Figure 7-7.
- ***The maximum battery capacity:*** Normally the capacity of the rechargeable battery is 2000 mAH (21600 Joule). Based on the power consumption requirement, the nodes should not cross the maximum capacity during the night (7 hours).

Based on the information provided by the designer, the code generator builds the topology configuration file and the simulator configuration file. The simulator configuration file contains the flags that were required to print the debugging statements and print the calls between the components, the nodes platform, the reporting CPU formats (i.e. cycles or seconds), and the number of nodes (see Figure 7-8).

```
#Each line should be a node like: nodeid x y z rho
#nodeid = node identifier
#x y z = 3D node coordinates
#rho: density of obstacles (if not present is zero) for each radio link
#simulator will take the maximum rho between transmitter and receiver
0   0 0   0 0
1   7  98  -4   0
2  5  62  -1  0
3  26  6  3  0
4  -5  131  -1  0
5  18  126  4  0
6  18  36  5  0
7  54  0   0  0
8  5  37  1  0
9   7  98  1  0
10  5  72  4  0
```

Figure 7-6: SensIV Topology Structure

Figure 7-7: Physical Location of SensIV Nodes

```
action=simulate
 colors=false
 banner=false
 platform=micaz
 update-node-id=true
 monitors=c-print, energy
 monitors=calls
 show-stack=false
 call-sites=false
 edge-types=false
 logfile=Node
 #report-seconds
 #real-time
 seconds-precision=1
 seconds=10000
 simulation=sensor-network
 nodecount=12
topology=./top.top
```

Figure 7-8: The Simulator Configuration

The generated code was instrumented by the debugging statements, which are necessary for the code validation procedure and the integration between the UML model and the simulator (see Appendix I for the generated code). Avrora simulates the code and prints out the debugging statements that display the code semantics as well as the signals between the modules, so that the designer can compare the semantics to the design semantics that are shown in the executed sequence diagram (see Figure 7-9and Figure 7-10). In Figure 7-9, the simulator log file displays the debugging statement output, which prints out the state-charts ID that caused the code generator to produce the executed code by the simulator. The designer can verify that the model semantics matches the code semantics through the following comparison:

- Comparison between the flow of the generated code semantics and the UML semantics: the sequence of the printed states-ID that was generated through debugging statements match the sequence of the states in the sequence diagram. For example, the debugging statements display the system flows as *BootHandler – Leds_S7-*

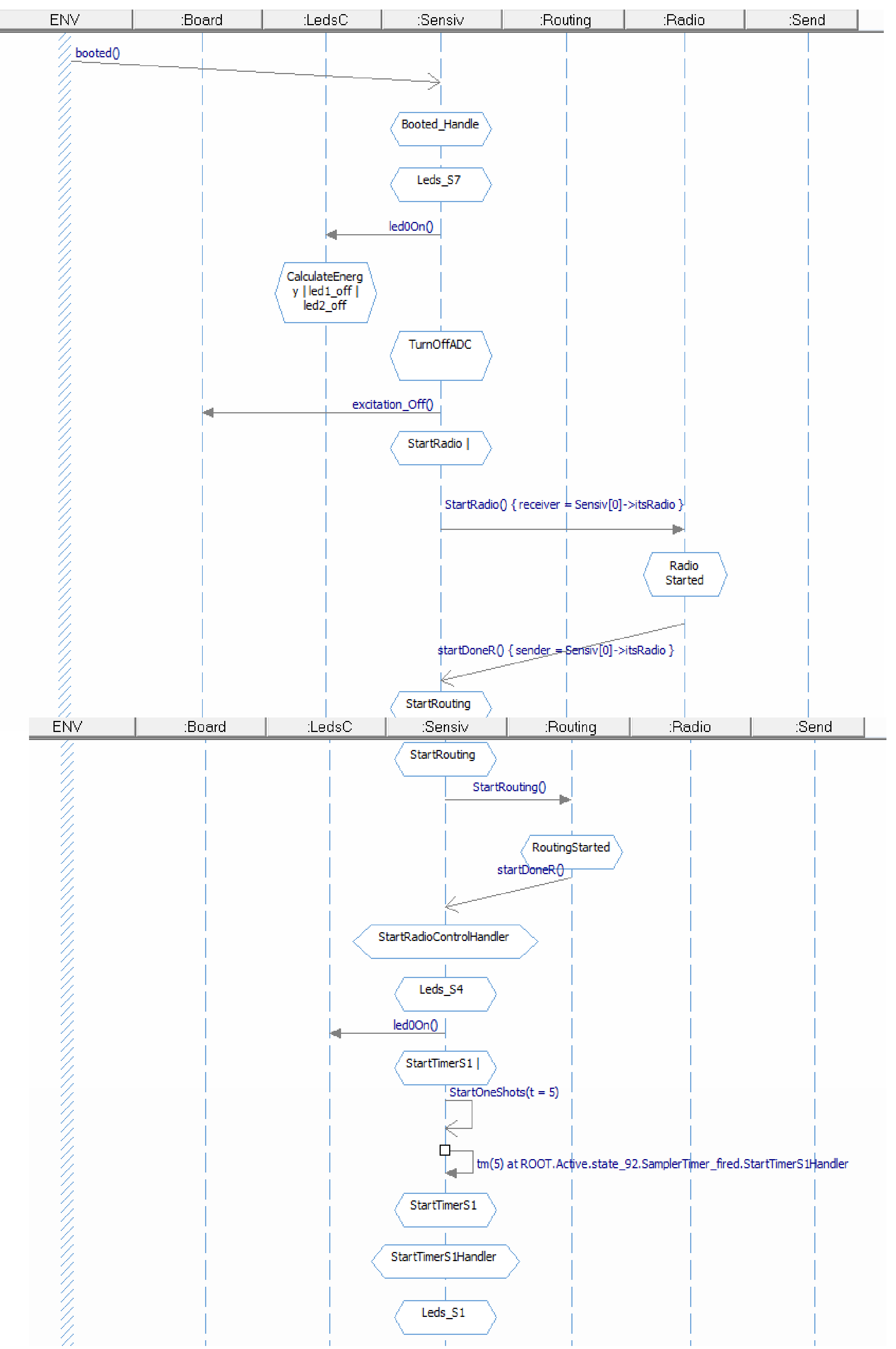
Turnoff ADC StartRadio (see Figure 7-9). The sequence diagram shows the exact same flow in the sequence diagram (see Figure 7-10).

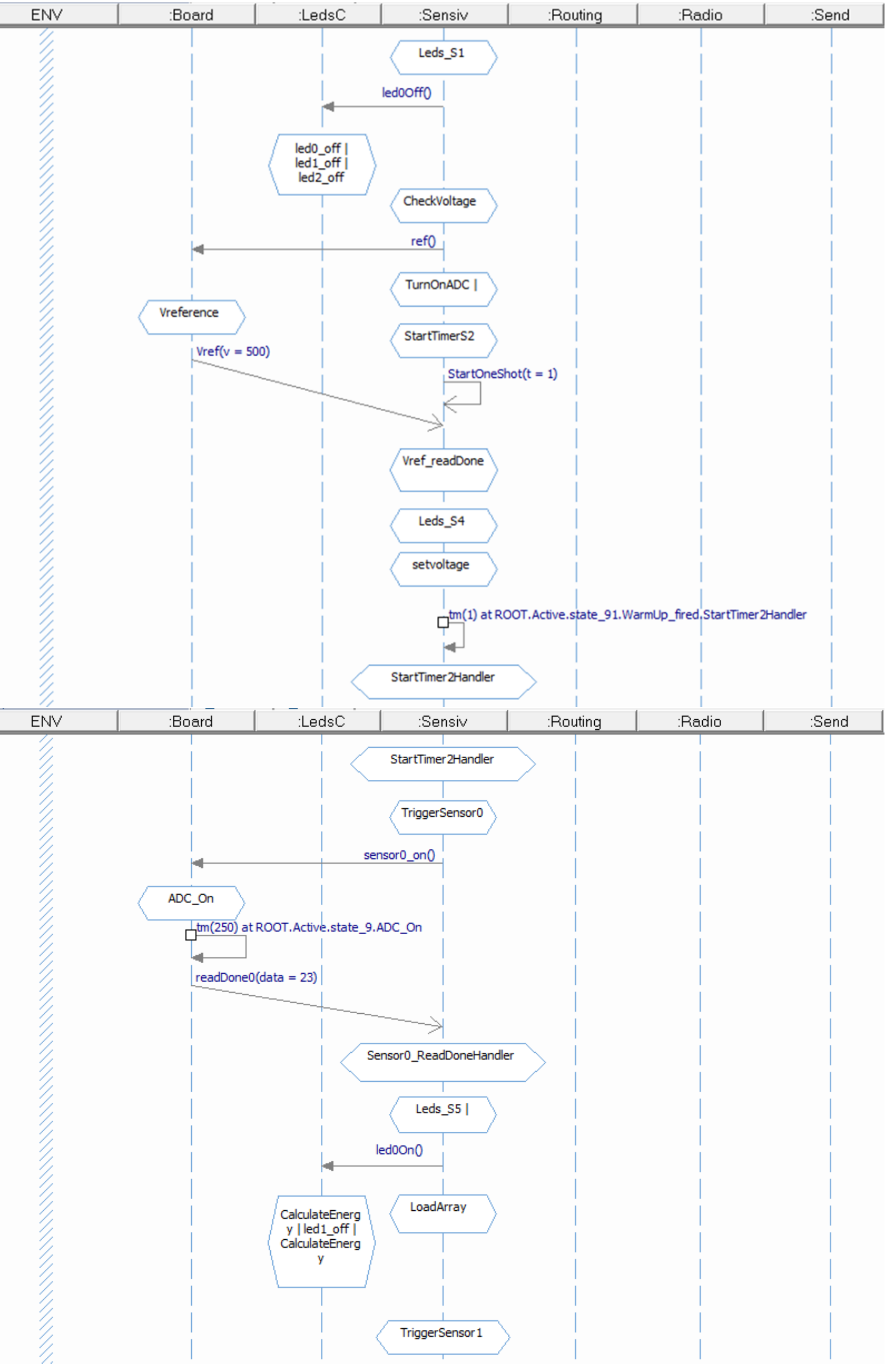
- Comparison between the signals and the OS components at the coding layer and the signaling between the UML classes: the simulator displays the signaling messages between the modules at a low level (detailed signals). Meantime, the UML sequence diagrams displays the exact message, but at a higher abstraction layer. For example, at the coding level, starting the radio is achieved by a signal generated by the application to the *CtpRoutingEngineP__0__StdControl__start* while this message is represented by *StartRadio* signal at the modeling layer.

```
0    8025990  |BootedHandler| St
0    8026078  |Leds_S7|
0    8026193  |TurnoffADC|
0    8026304  |StartRadio|
0    8026306      --> PowerCycleP__SplitControl__start
0    8026464      <-- PowerCycleP__SplitControl__start
0    8026585  |StartRouting|
0    8026592      --> CtpRoutingEngineP__0__StdControl__start
0    8026621          --> CtpRoutingEngineP__0__chooseAdvertiseTime
0    8028854          <-- VirtualizeTimerC__0__startTimer
0    8028866      <-- CtpRoutingEngineP__0__StdControl__start
0    8029015  |BootedHandler| En
0    8029048      --> SchedulerBasicP__TaskBasic__runTask
0    8029104          --> PowerCycleP__startRadio__runTask
0    8029108              --> CC2420CsmaP__SplitControl__start
0    8045556  |StartRadioControlHandler| St
0    8045644  |Leds_S4|
0    8045773  |StartTimerS1|
0    8045775                  --> Atm128AlarmAsyncP__0__Counter__get
0    8045807                  <-- Atm128AlarmAsyncP__0__Counter__get
0    8045815                  --> VirtualizeTimerC__0__startTimer
0    8045901                  <-- VirtualizeTimerC__0__startTimer
0    8046120  |StartRadioControlHandler| En
0   43881941          <-- Atm128AlarmAsyncP__0__Counter__get
0   43881943          --> VirtualizeTimerC__0__fireTimers
0   43882404  |StartTimerS1Handler| St
0   43882492  |Leds_S1|
0   43882614  |ReadVoltage|
0   43882617              --> ArbiterP__1__Resource__request
0   43882646                  --> SchedulerBasicP__TaskBasic__postTask
0   43882690                  <-- SchedulerBasicP__TaskBasic__postTask
0   43882704              <-- ArbiterP__1__Resource__request
0   43882804  |TurnOnADC|
0   43882935  |StartTimerS2|
0   43882937              --> Atm128AlarmAsyncP__0__Counter__get
0   43882969              <-- Atm128AlarmAsyncP__0__Counter__get
0   43883267  |StartTimerS1Handler| En
0   43950594  |Vref_readDone Handler| St
0   43950682  |Leds_S4|
0   43950897  |Vref_readDone Handler| En
```

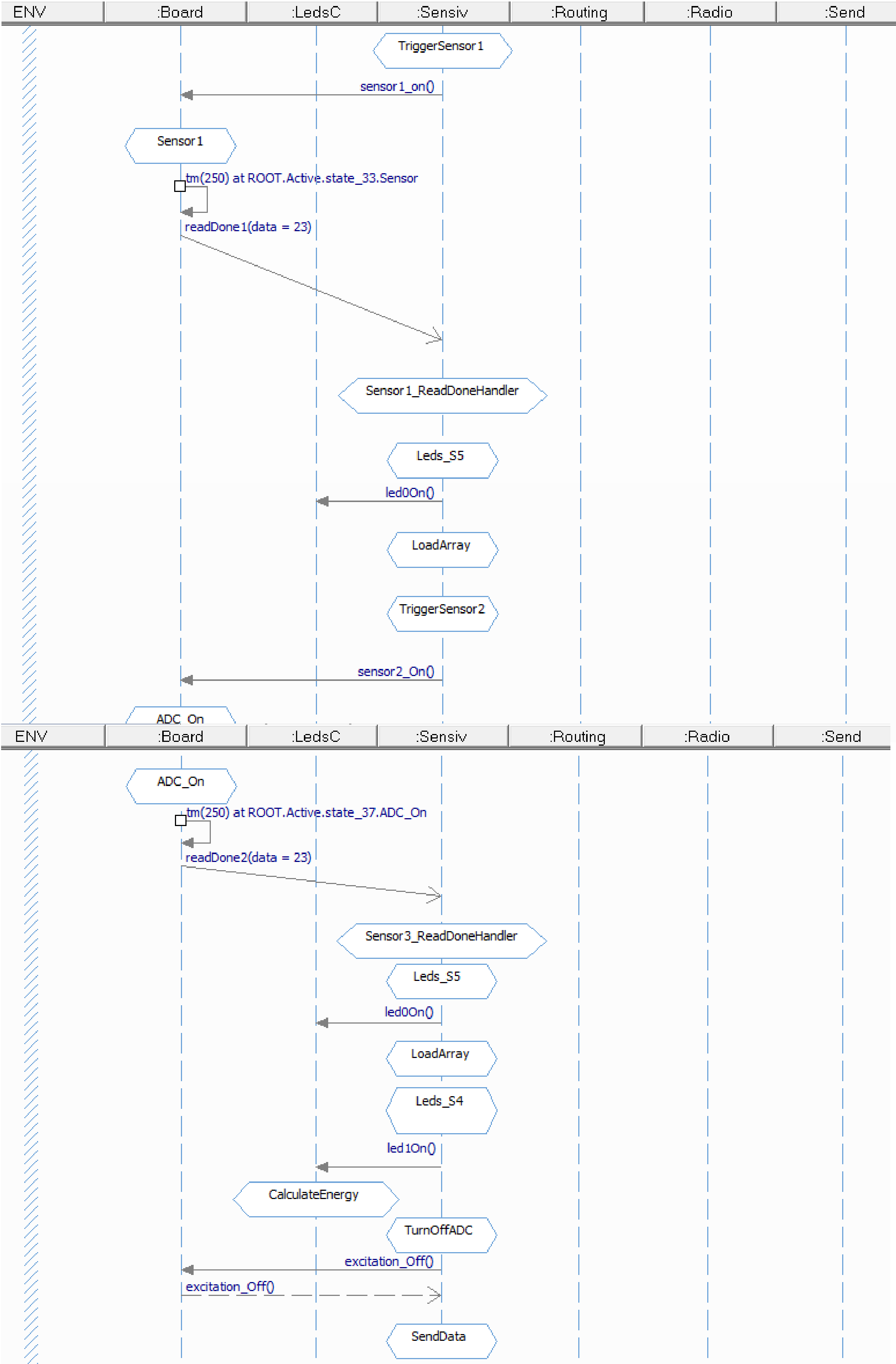
```
0   51039263  |StartTimerS2TimerHandler| St
0   51039393  |TriggerSensor0|
0   51039396              --> ArbiterP__1__Resource__request
0   51039425                  --> SchedulerBasicP__TaskBasic__postTask
0   51039469                  <-- SchedulerBasicP__TaskBasic__postTask
0   51039483              <-- ArbiterP__1__Resource__request
0   51039667  |StartTimerS2andler| En
0   51106635     |Sensor0_readDoneHandler| St
0   51106723  |Leds_S5|
0   51106831  |LoadArray|
0   51106961  |TriggerSensor1
0   51106964          --> ArbiterP__1__Resource__request
0   51107007          <-- ArbiterP__1__Resource__request
0   51107172  |Sensor0readDone| En
0   51171422  |Sensor1readDoneHanlder| St
0   51171510  |Leds_S5|
0   51171646  |TriggerSensor2|
0   51171649          --> ArbiterP__1__Resource__request
0   51171692          <-- ArbiterP__1__Resource__request
0   51171904  |Sensor1readDoneHanlder| En
0   51236231  |Sensor2readDoneHandler| St
0   51236319  |Leds_S5|
0   51236455  |TriggerSensor2|
0   51236458          --> ArbiterP__1__Resource__request
0   51236501          <-- ArbiterP__1__Resource__request
0   51236715  |Sensor2readDoneHandler| En
0   51301036  |Sensor3readDoneHandler| St
0   51301124  |Leds_S5|
0   51301239  |TurnoffADC|
0   51301642  |SendData|
0   51301650              --> CC2420ActiveMessageP__AMSend__send
0   15193363              <-- CC2420ActiveMessageP__AMSend__send
0   51301670  |Sensor3readDoneHandler| En
```

Figure 7-9: The Call Trace Produced by Avrora


ENV
:Board
:LedsC
:Sensiv
:Routing
:Radio
:Send
booted()
Booted_Handle
Leds_S7
led0On()
CalculateEnerg y | led1_off | led2_off
TurnOffADC
excitation_Off()
StartRadio |
StartRadio() { receiver = Sensiv[0]->itsRadio }
Radio Started
startDoneR() { sender = Sensiv[0]->itsRadio }
StartRouting
StartRouting
StartRouting()
RoutingStarted
startDoneR()
StartRadioControlHandler
Leds_S4
led0On()
StartTimerS1 |
StartOneShots(t = 5)
tm(5) at ROOT.Active.state_92.SamplerTimer_fired.StartTimerS1Handler
StartTimerS1
StartTimerS1Handler
Leds_S1



ENV
:Board
:LedsC
:Sensiv
:Routing
:Radio
:Send
Leds_S1
led0Off()
led0_off | led1_off | led2_off
CheckVoltage
ref()
TurnOnADC |
Vreference
StartTimerS2
Vref(v = 500)
StartOneShot(t = 1)
Vref_readDone
Leds_S4
setvoltage
tm(1) at ROOT.Active.state_91.WarmUp_fired.StartTimer2Handler
StartTimer2Handler
ENV
:Board
:LedsC
:Sensiv
:Routing
:Radio
:Send
StartTimer2Handler
TriggerSensor0
sensor0_on()
ADC_On
tm(250) at ROOT.Active.state_9.ADC_On
readDone0(data = 23)
Sensor0_ReadDoneHandler
Leds_S5 |
led0On()
CalculateEnergy | led1_off | CalculateEnergy
LoadArray
TriggerSensor1



ENV
:Board
:LedsC
:Sensiv
:Routing
:Radio
:Send
TriggerSensor1
sensor1_on()
Sensor1
tm(250) at ROOT.Active.state_33.Sensor
readDone1(data = 23)
Sensor1_ReadDoneHandler
Leds_S5
led0On()
LoadArray
TriggerSensor2
sensor2_On()
ENV
:Board
:LedsC
:Sensiv
:Routing
:Radio
:Send
ADC_On
tm(250) at ROOT.Active.state_37.ADC_On
readDone2(data = 23)
Sensor3_ReadDoneHandler
Leds_S5
led0On()
LoadArray
Leds_S4
led1On()
CalculateEnergy
TurnOffADC
excitation_Off()
excitation_Off()
SendData

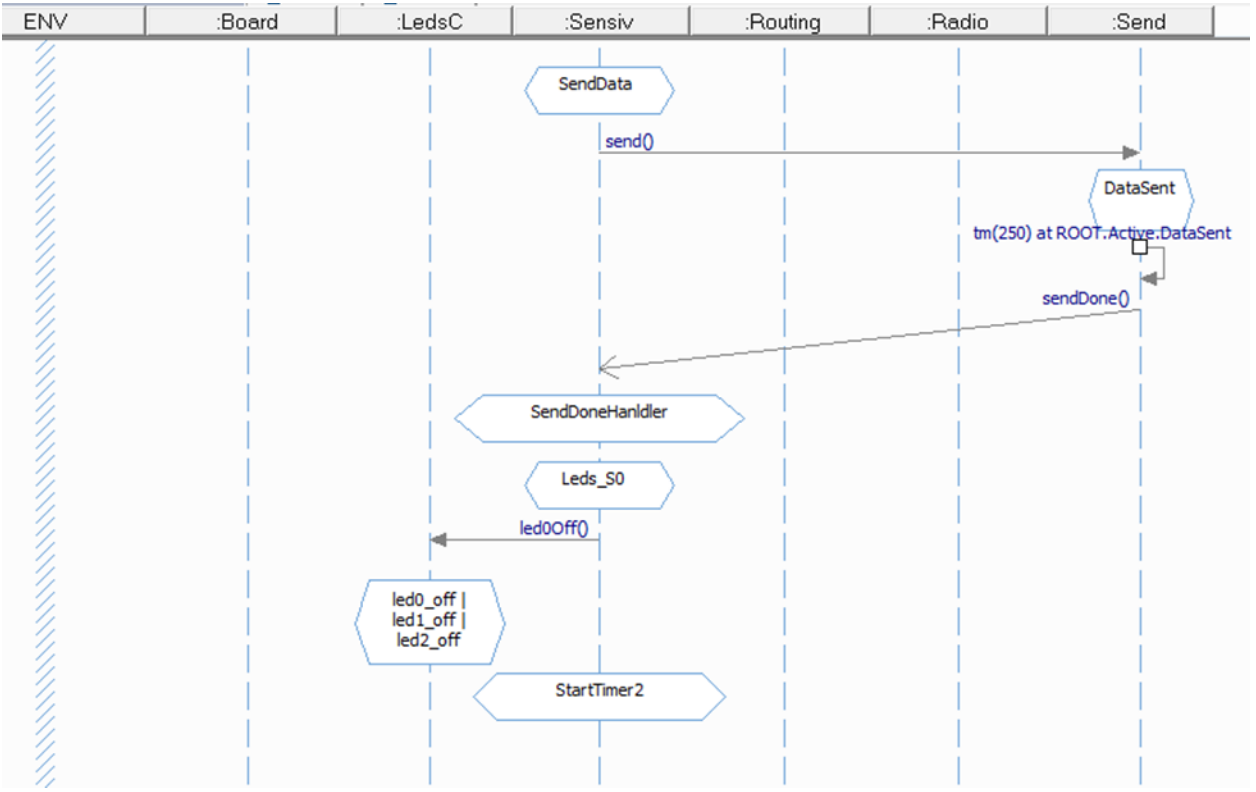


Figure 7-10: SensIV Executed Sequence Diagram

For the power consumption, Avrora generates the following files:

- ***Power log file:*** The power log file contains the power consumed by each hardware resource together with the CPU cycle. The file is updated whenever one of the resources is signaled (see Appendix II for the log file structure).
- ***Trace file:*** The trace file contains the CPU cycles for each operation (see Appendix III for the trace file structure).

In order to demonstrate how the power consumption report can enable the designer to improve the power consumption of SensIV system, an initial analysis was run for SensIV. The analysis for SensIV shows that the highest amount of power is consumed by nodes 1, 6, 8, and 9 and the power consumed was 7289 Joules while the lowest amount of power was consumed by the other nodes and power consumed was 4368 Joules (see Table 7-4). Therefore, the total amount of power consumed by the worst nodes for 7 hours was 7289 Joules for the communication process (analyzed by the simulator) plus 350 Joules for the deterministic

components (analyzed by the executed model) which equals 7639 Joule total. This total is less than 21600 Joule which is the battery's maximum capacity which implies that the nodes will not die before the battery is recharged. The power consumption for the nodes 1, 6, 8, and 9 are considered the worst power consumption and is shown in Table 7-4.

Node ID	Total Power Consumption (Joules)
0	4368
1	7289
2	4368
3	4368
4	4368
5	4368
6	7289
7	4368
8	7289
9	7289

Table 7-3: SensIV Nodes Total Power Consumption

Event Handler	**Signaling Rate**	**Breakdown Power Consumption for Hardware Resources (Joule)**						
		CPU	Radio	Board	Green LED	Yellow LED	Red LED	Total
|TimerS1Handler|	8617	Energy Neglected						
|SendDoneHandler|	8617							
|StartDoneHandler|	1							
|RoutingStart|	1	0.052	0.019	0.0056	0	0	0	0.775
|CallSend|	8617	1139.638	3132.180	146.104	228.193	228.193	228.193	5101.501
|Booted|	1	Energy Neglected						
|ReceiveHandler|	8617	509.222	1536.031	57.190	33.110	33.124	18.921	2187.598

Table 7-4: Power Consumption Report for Node 1, 6, 8, and 9

As explained in Chapter 6, the parsing tool provides the designer with the power consumption breakdown report that contains the following information:

- The power consumption for each hardware resources activity (i.e. CPU, Radio, LEDs, and Board) that was associated with each event handler and each operation call.
- The rate of signaling for each event handler and each operation.

The information provided by the parsing tool gives the designer the following alternatives in order to improve the power consumption of the design:

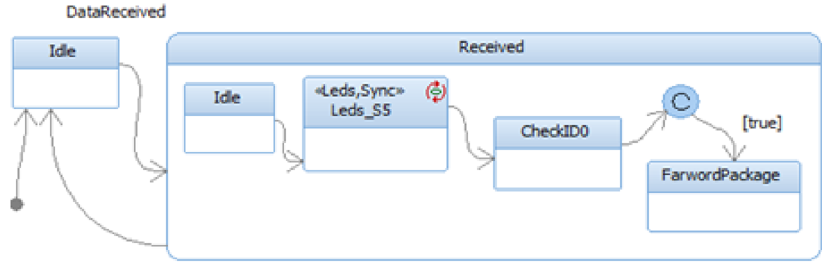
- Modifying the Software Logic
  - Change Event Handler Structure
  - Switching Data Collection Protocol
- Changing Network Configuration
  - Low power Listening Mode

7.3.4.1 Modifying the Software Logic

7.3.4.1.1 Modifying Event Handler Structure

In case one of the event handlers is signaled often, the designer can change the event handler structure in order to decrease the calls to the hardware resources that were located in the event handler and eventually the power consumption will be decreased. For example, based on Table 7-4 the event handlers |*ReceiveHandler*| is signaled 8617 times once a data package is received. If the designer removes the calls for the LEDs from those particular event handlers, the power consumption of the event handler will decrease and ultimately, the power consumption of the node will decrease.

The modifications of the event handlers were developed in the UML state-chart. The code generator considers the change once the code is produced. Figure 7 9-A shows the state-chart while the calls for the LEDs were included in event handler structure while Figure 7 9-B shows after the event handler is modified to remove the calls to the LEDs. The UML model modification leads to the remove the code associated with LED_S5 state. These modifications have enhanced the power consumption for the nodes 1, 6, 8, and 9 as shown in Table 7-5.



A. Receive Hander Before Design Modification

B. Receive Handler After Design Modification

Figure 7-11: Design Modification for Event Structure

Event Handler	**Signaling Rate**	**Breakdown Power Consumption for Hardware Resources (Joule)**						
		CPU	Radio	Board	Green	Yellow	Red	Total
|TimerS1Handler|	8617	Energy Neglected						
|SendDoneHandler|	8617							
|StartDoneHandler|	1							
|RoutingStart|	1	0.052	0.019	0.0056	0	0	0	0.775
|CallSend|	8617	1139.638	3132.180	146.104	228.193	228.193	228.193	5101.501
|Booted|	1	Energy Neglected						
|ReceiveHandler|	8650	509.222	1536.031	0	0	0	0	2045

Table 7-5: Power Consumption for Nodes 1, 6, 8, and 9 after Modification

The removal of the LEDs calls from the event handlers is a simple example to show the alternatives that the design can do to improve the energy consumption by modifying the event handlers' structure. However, the example shows that importance of the network simulation to determine the signaling rate of event handlers that are related to the network behaviour. Therefore, the network simulation results and the parsing for simulator results can be a guideline to the designer to change the design of the event handlers at the modeling layer.

7.3.4.1.2 Switching Data Collection Methodology

The initial analysis of SensIV power consumption indicated that the sending operation was called 8617 and consumed 5101 Joule which is around 70% of the total consumed energy. Changing the rate of calling the send operation definitely improves the amount of energy consumed by this operation.

There are two common data collection methodologies for WSN; aggregation and non-aggregation protocols. In the non-aggregation methodology, the data packets are transmitted without any node holding the packets for processing. This technique leads to call the sending operation often which leads to increase the radio power of the communication process among the network. In the data aggregation methodology, the node does not send the sensed data directly after sensing process is completed. However, the node waits to a specific event to occur and then combine both the sensed data and the event data in one data packet and send both data packets together. For example, the parent node waits until it receives the data from the child node and combines both the generated data and the child node data and send the data in one package. Another example, the node senses the temperature every cycle and sends the average temperature of two cycles. Sending the average of the two temperatures instead of sending the data after every sensed cycle will decrease the amount of power consumed in the communication process among the network [38].

The initial design of SensIV was designed to use the non-aggregation methodology since each node sends the data directly after the sensing process is completed. SensIV software component was modified to use aggregation technique by averaging the two temperature values that were produced every two cycles. The send function is called after the average is calculated

to send the average value. This lead to decrease the number of calls to the “send” operation and the amount of powered consumed for the communication process.

As shown in Figure 7-12, the state-chart is modified to calculate the average between the current sensed temperature and the previous sensed temperature. The sending operation is called every 6 minutes to send the average temperature of two values since the node is activated every 3 minutes to sense the temperature.

The design modification that changes the design from non-aggregation to aggregation data collection technique leads to decrease the number of calls for the sending operation from 8677 calls to 4305 (see Table 7-5 and Table 7-6). Moreover, since the number packets sent by each child node are decreased, the number of packets received by the parent nodes is decreased as well. Decreasing the amount of data packets that were transferred across the network, leads to decreasing the amount of power consumption in the communication process. Figure 7-13 and Figure 7-14 compare the power consumed for sending and receiving by each node for both data collection protocols, aggregation protocol and non-aggregation methodologies.

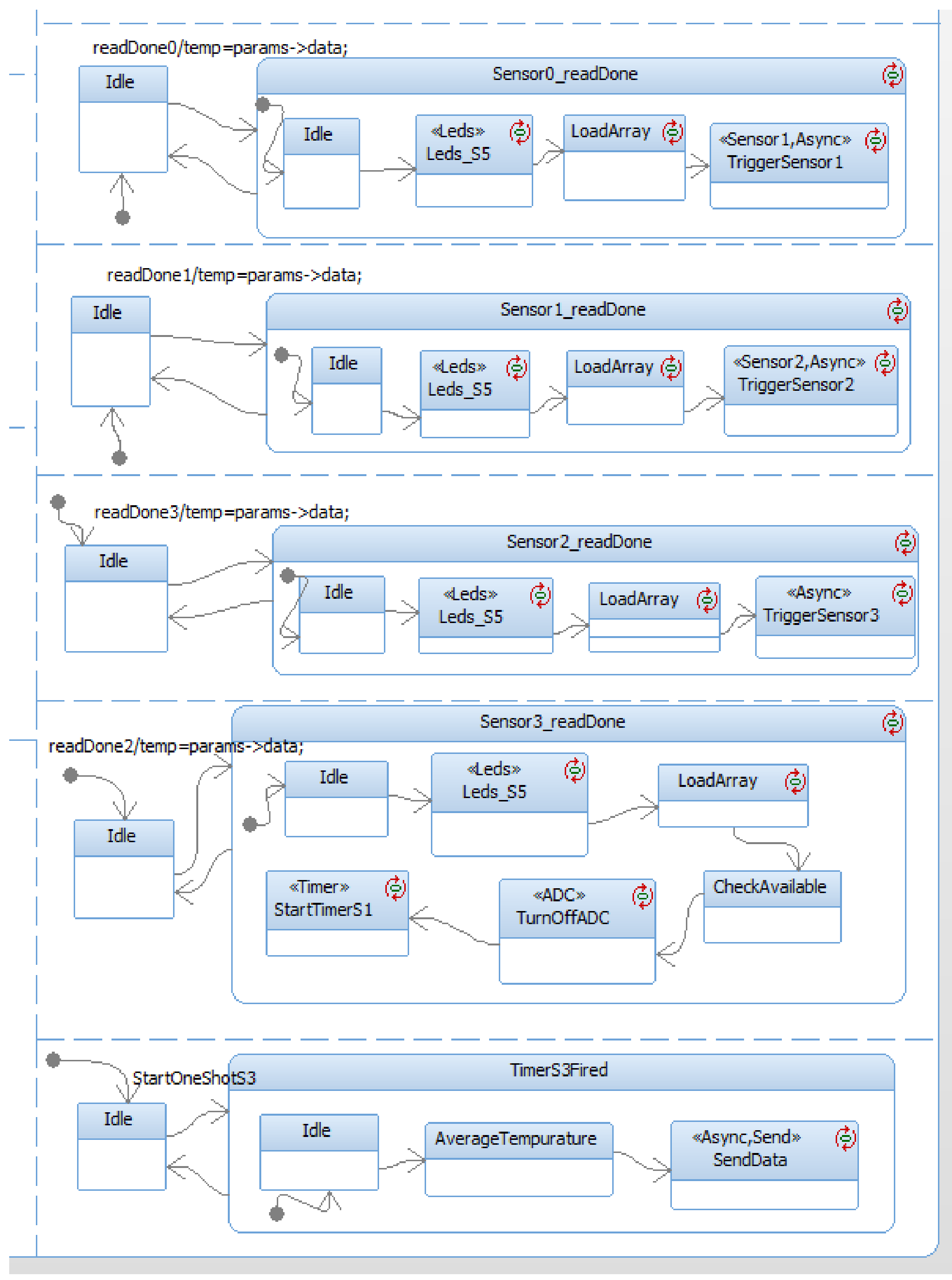


Figure 7-12:Aggregation SensIV State-Chart Design

Event Handler	Signaling Rate	Breakdown Power Consumption for Hardware Resources (Joule)						
		CPU	Radio	Board	Green	Yellow	Red	Total
|TimerS1Handler|	8617	Energy Neglected						
|SendDoneHandler|	8617							
|StartDoneHandler|	1							
|RoutingStart|	1	0.052	0.019	0.0056	0	0	0	0.775
|CallSend|	4305	569.261	1565.634	73.045	0	0	0	2207.94
|Booted|	1	Energy Neglected						
|ReceiveHandler|	4305	140.9	485.604	18.06	0	0	0	2045

Table 7-6: Aggregation Power Consumption for Nodes1, 6, 8, and 9

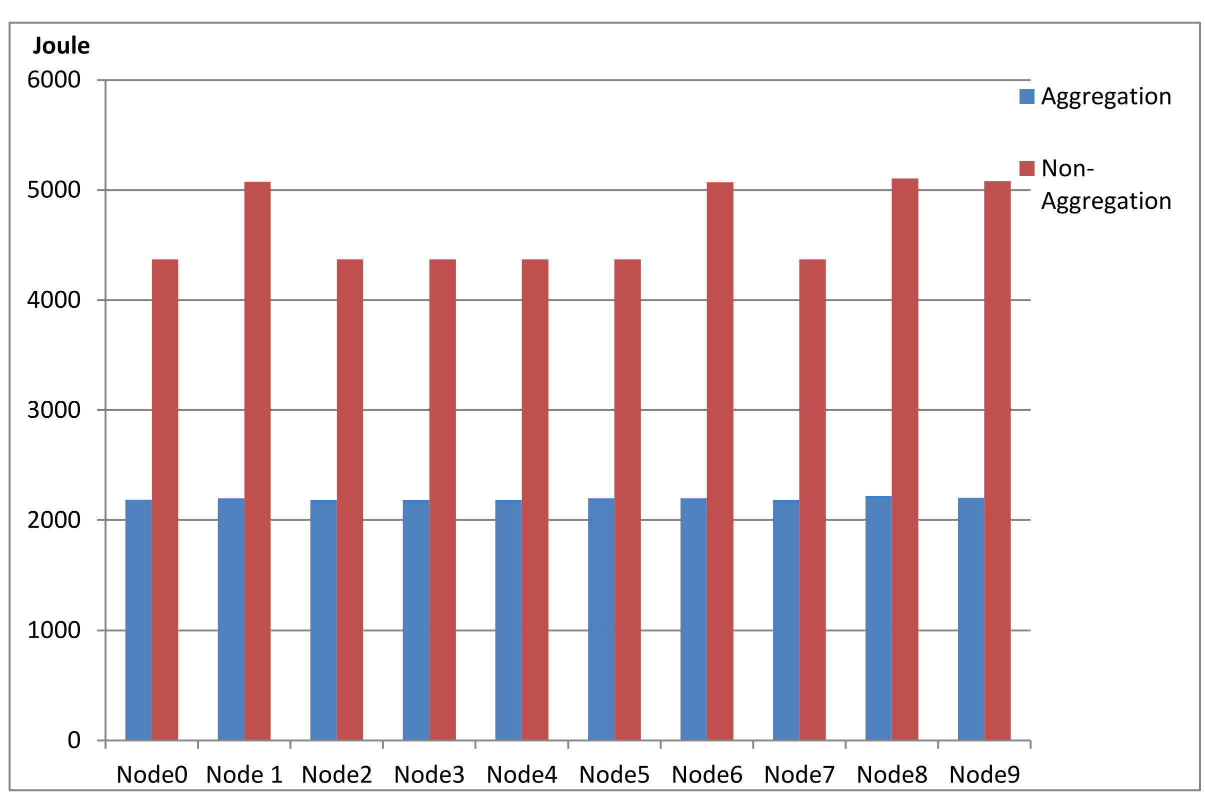


Figure 7-13: Power Consumption of Network Sending Process

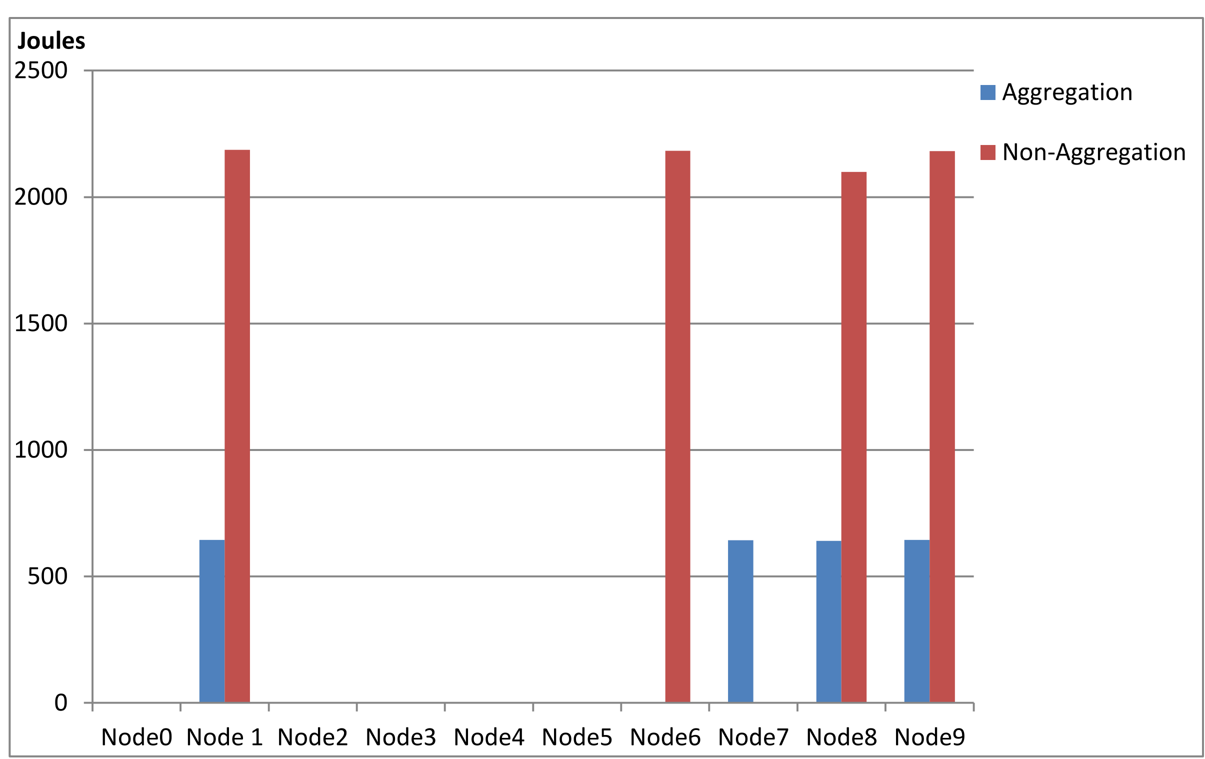


Figure 7-14: Power Consumption for the Network Receive Processing Process

7.3.4.2 Network Configuration

7.3.4.2.1 Low Power Listening Mode

TinyOS offers a low power listening mode (LPL) that decreases the amount of power consumed by radio. LPL mode wakes up the node through time intervals to detect if the transmitter node sends any packets. The transmitter node keeps sending the packets until it receives acknowledgments or timeout from the linked layer. Once the receiver node receives the first data packet, the node stay awake until the sender receives the rest of the packets then it goes to sleep mode and wake up again at the end of the time interval.

The LPL is a design choice that should be enabled once the node the node is booted. The LPL is enabled through the code line *call myLPL.setLocalSleepInterval(LPL_INTERVAL),* where *myLPL* is an instance of the wiring component. The implementation of LPL is offered by TinyOS support component which manages all the logic required for the mode. In order to enable the

mode the only change required to the design will be adding a state "*Enable LPL_300*" in the state-chart as shown in Figure 7-15. The code generation will produce the necessary component wiring and the required code.

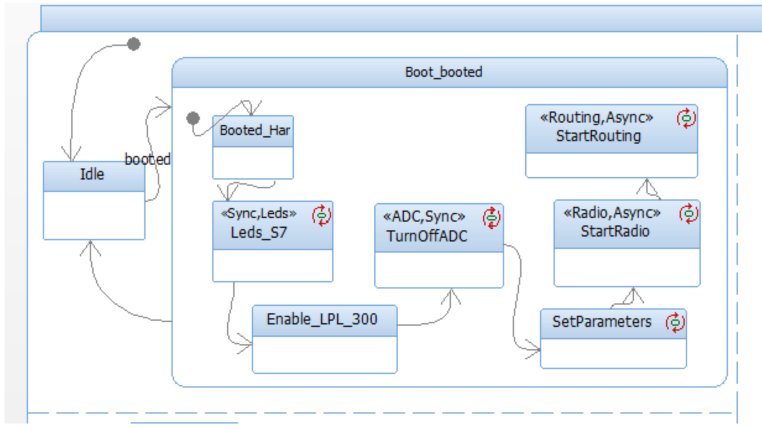


Figure 7-15: Enabling LPL in Booted Event

Enabling the LPL saves significant amount of power that was consumed for the receiving process since the radio unit of the receiving node is not open for a long time to receive the package compared to the non-LPL mode. Figure 7-16 compares the power consumed by the receiving process for all nodes. As it shown, in the figure the amount of power saved is around 50% compared to the normal network operation (non-LPL).

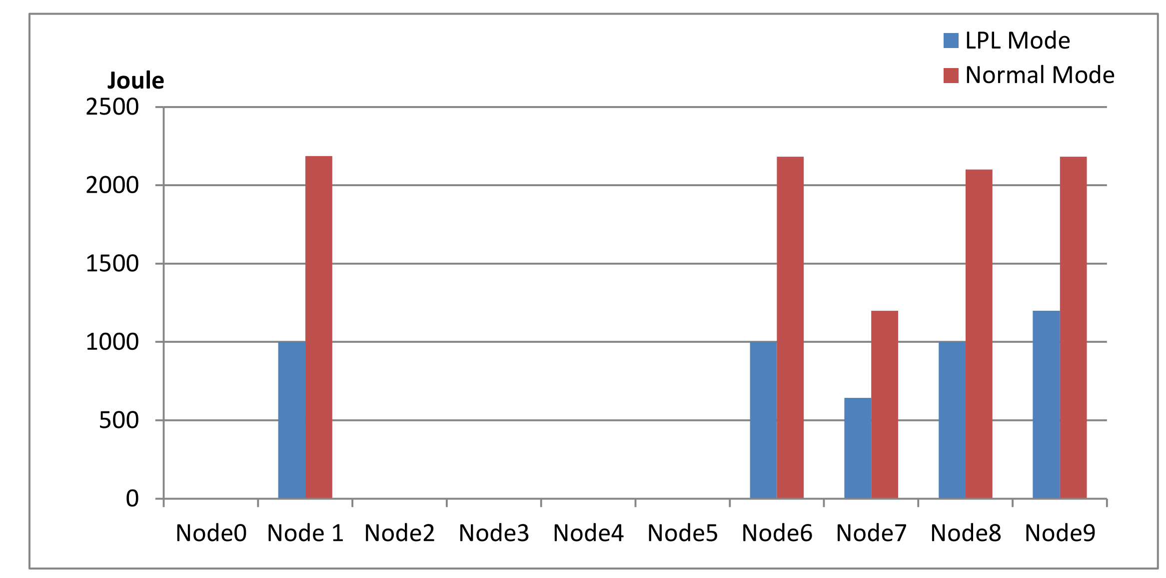


Figure 7-16: Power Consumption of Receiving Energy for LPL Vs Normal Mode

7.3.5 Network Analysis Accuracy

The focus of this thesis is the development of a framework and of a methodology to design and analyzes the power consumption of WSN. The accuracy of the analysis values is not the focus of this thesis. Nevertheless, the analysis results reflect the impact of the design enhancements of the overall power consumption. The analysis results can indicate to the designer whether the design enhancements have improved the overall power consumption of the system.

Additionally, the accuracy of the power consumption analysis is driven by the accuracy of the energy model (i.e. the annotated power model values):

- The power consumption values that are annotated in the UML model and are used by the model execution technique to analyse the power consumption of the hardware component, are measured while the actual hardware system was running (see Section 4.5.1).
- The power consumption values that are used by Avrora simulator to analyze the network power consumption are calculated by measuring the power consumption of an actual hardware system while the system was running. Based on the publication of Avrora development team that were published in [39], the error of the power consumption analysis of the simulator compared to the power consumption of the actual hardware was measured using an Oscilloscope is 0.4% with standard deviation: 0.24.

7.4 Framework Automation

As the previous chapters show, the design process includes the following:

- The design representation using the UML design patterns
- The requirement validation using the sequence diagram comparison feature

- The power analysis at the modeling layer for the hardware component
- The code and the simulator configuration generation for the platform
- The network simulation and the power consumption analysis for the network
- The simulation results parsing to integrate the results with the UML models

Table 7-7 shows the automated intervention and the designer intervention of each stage of the process.

Stage	Designer Intervention	Automated Intervention
Representing the design in the UML layer	• The designer uses the UML design patterns to represent the design structure and behaviour • The designer initiates the XML toolkit	• XML tool kit provided by Rational Rhapsody software generates the XML text file that can be parsed by the code generator
Analysis at the modeling layer	• Using the sequence diagram comparison results to correct the design error • Using the power consumption results to enhance the design	• Compares the sequence diagrams of the multiple scenarios and uses the colour code to indicate the missing signals, timing values, etc. • Calculates the power consumption of the hardware component
Code generation	• Imports the XML file to the code generator • Inputs the network topology structure • Inputs some of the platform specification, such as battery capacity and node platform	• Builds the target code • Builds the simulator configuration • Instruments the code with the necessary debugging statements
Simulations	• Imports the generated code • Imports the simulation configuration file • Runs the simulator	• Simulates the network behaviour • Calculates the power consumption for each node and generates the log file • Generates the trace file for

		the debugging statements
UML model and simulator integration	• Imports the trace file and the logs file to the parsing tool • Uses the values printed by the parsing tool to determine the enhancements to the model	• Parses the simulator files • Calculates the power consumption of each event, event handler, and method and prints the values to the designer

Table 7-7: Designer and Automated Intervention Comparison

To summarize, the designer intervention is as the following:

- Linking the inputs and outputs of each stage
- Interrupting the analysis results in order to modify the design

To summarize, the automated intervention is as the following:

- Executing the design model
- Generating the required code and the simulator configuration files
- Simulating the network behaviour
- Parsing the simulation results and displaying the power consumption results to the designer

7.5 Framework Generality

The tools that were leveraged to developed this thesis approach can be classified as the developed tools as part of the thesis contribution (i.e. UML models and code generator) and the tools that were leveraged to develop the models and simulate the network behaviour (i.e. Rational Rhapsody and Avrora Simulator) as introduced in Section 1.4. This section discusses the generality of the thesis tools, such as the support for other operating systems other than the case study operating system TinyOS, using alternative UML tools to develop the models, and using alternative simulators to simulate the network behaviour.

7.5.1 UML Model and Code Generator Generality

The design patterns were inspired by a field design called SensIV, which is TinyOS based operating system. However, the patterns can be leveraged to represent the design of other WSN operating systems that have similar design semantics to TinyOS, such as Contiki [15]. Contiki operating system offers a group of OS components that contain a group of methods and parameters to control the hardware elements of the node and those components can be represented by the service component patterns (see Section 4.3.1.). In addition, Contiki operating systems is an event based operating systems, therefore, using the event handler's pattern will capture the flow of the design (see Section 4.3.2.).

Moreover, the syntax that defines the hardware components, such as ADC, Radio, and Leds, are defined such that the syntax is generic so that the designer can use the same model for generating the proper code for multiple WSN platforms.

The design behaviour is represented in the model using the events and the event handlers. The code generator parses the XML representation and creates a vector of the states in each event handler and loads the code associated with each state from the database (see Section 5.3). The basic requirements for the code generator to successfully generate the code is the event based nature of the design and the correct code syntax in the database. Therefore, the code generator is capable of generating the code for any event-based system. Also, the instrumentation strategy relies on inserting debugging statements at the beginning of each event handler and the calls of each methods. Those characteristics are common among any event-based system.

7.5.2 The Generality of the Leveraged Tools

The developed framework relies on the mentioned tools, Rational Rhapsody and Avrora simulators, to create the UML model and the simulate the network behaviour respectively. Other UML tools can be used instead of Rational Rhapsody to develop the UML models. However, the alternative UML tool has to be capable of generating the XML representation of the UML design. Since the XML file structure is standard, the introduced code generator in this thesis is capable of parsing the XML file and generating the proper code. In case the alternative UML tool does not support model execution technique, then the designer has to rely on the simulator completely to analyze the power consumption of the hardware components instead of using the model execution technique in Rational Rhapsody to analyze power consumption of the hardware components in the modeling layer.

An alternative for Avrora simulator is Tossim [36] simulator. Both simulators require the nesC code to analyze the power consumption of the hardware components and the network behaviour and both simulators support code instrumentation. Therefore, using Tossim simulator will require changing the debugging statements inserted by the code generator to instrument the code, for instance from *printStr* (used by Avrora) to *db* (used by Tossim), while the printed string will remain the same. However, Avrora simulator was chosen to conduct the power consumption analysis for the network due to the detailed log file for the power consumption. The entries in Avrora log file are updated with the power drained from battery, as well as, the time stamp of the process. The log file details are essential for the parsing tool to integrate the simulation results with the UML models.

7.6 Threats to Validity

This section discusses the WSN system ontology in order to explore the capability of the developed framework to design and to analyze various types of WSN systems. A typical WSN system consists of a group of nodes, which contain a sending and receive unit, a group of attached sensors, a processor, memory, and a deployed software application. In addition, the WSN system contains a routing protocol, a topology structure for the nodes, and a gateway node that collects all the data. The nodes communicate wirelessly to deliver the information to one gateway node that has sending and receiving capabilities. Normally, the gateway is attached to central computer to further process the collected data and enable global access to the data through internet. Based on AKyildize et. Al. [6] and Yick et. Al. [40] WSN systems can be classified based on:

7.6.1 Monitored Data Type

Examples of monitored data include temperature, humidity, and pressure. One WSN node can contain multiple sensors of the same type, for example multiple temperature sensors or multiple sensor types, such as a temperature and a pressure sensor. From a power consumption efficient design perspective, the type of the monitored phenomena is irreverent to the power consumption analysis compared to the power consumed by the sensor hardware. The power consumption analysis framework that was developed in this thesis requires annotating the power consumed by the sensor such that framework can analyze the overall power consumption of the sensor hardware.

7.6.2 Data Collection Protocols

The WSN data collection protocols can be categorized as follows:

- *Point-to point protocol*: The nodes sense the phenomena and send the sensed data to the gateway node directly without passing the data packet through any neighbour nodes. The communication process can occur through optical signals, such as the system developed in [41] or occur through wireless signal that follows 802.11 protocol, such as the system developed in [42]. The communication process has a deterministic nature since there is no data forwarding or control packets involved (see Section 2.2). Therefore, the power consumption of the communication process can be analyzed through annotating the UML model with the sending and receiving power values. The power consumption can be analyzed using the model execution techniques that were used to analyze the power consumption of the SensIV hardware component (see Section 7.3.3) since the hardware components in both cases have a similar operating scenario.
- *Multi-hop communication protocol*: The routing protocol builds a routing tree that enables the nodes to pass the data between multiple nodes until the data is delivered to the gateway. Using the multi-hop protocol enables the designer to deploy multiple nodes and so increase the monitoring resolution to cover the operating fields using limited wireless range nodes. This thesis framework is capable of analyzing the power consumption of the multi-hop communication protocol, which was illustrated through power consumption analysis for SensIV that uses CTP protocol (see Section 7.3.4).

7.6.3 Fault Tolerance Capability

Some of the sensors may fail while they are operating due to power consumption issues or destruction of the sensor caused by the operating conditions, such as the military WSN

systems. The capability of the sensors to function without interruption due to the failure relies primarily on the software logic that controls the nodes behaviour. For instance, if one of the nodes dies the routing re-establishes the network routing tree to avoid the dead node.

This thesis presented a framework that analyzes the influence of changing the software logic on the network power consumption. The WSN simulator simulates the network behaviour and considers the software logic modification that occurred at the model level such that the simulator analysis results reflect the impact of these modifications (see Section 7.3.4).

7.6.4 Hardware Constraints

The hardware constraints of a WSN node are defined as the sensing unit, the radio unit, the processor unit, and the memory unit. The WSN systems are classified based on the hardware constraints as follows:

- The cost of the hardware constraints.
- The capability of the hardware constraints to operate in high densities.
- The hardware constraints capability to adapt to the environment.
- The power consumption of the hardware constraints.

The focus of this thesis is the designing of power aware WSN systems. The power consumption of the hardware constraints can be predicted using the UML modeling analysis method introduced in Section 4.5.1 and are demonstrated by the analysis for SensIV hardware components (see section7.3.3).

7.6.5 Network Topology

The network topology defines the physical location of the network nodes. Defining the topology during the designing phase reduces the installation cost and eliminates the need to re-organize the topology. This thesis does not focus on designing the optimum topology of the

network. However, the topology structure influences the worst node for power consumption since the node's location determines the amount of packets that are passed through that particular node. The framework enables the designer to insert the network topology to the code generator. The code generator produces the simulator configuration that includes a network topology (see Chapter 5). Based on the network topology, the network simulator analyzes the power consumption of each node considering the topology structure. Using the parsing tool, this thesis framework integrates the simulation results with the UML model (see Section 6.3). The designer can test the influence of the different topology configurations on the overall power consumption.

7.6.6 Software Design

The software design controls the hardware components of the nodes and the node behaviour. Based on the survey mentioned above, the software design can be categorized into 2 main categories:

- *Sensor Management Protocol (SMP):* The sensor software logic provides software operations and methods to control the data aggregation, exchanging data to locate the sensor, time synchronization, and turning on/off hardware components. The SMP structure influences the power consumption of the nodes. This thesis examined the impact of modifying the sensor software logic on the power consumption through using the model execution techniques and the integration of using the simulation with UML models. Sections 7.3.3 and 7.3.4 demonstrate the framework capability to detect the impact of turning the SensIV hardware components on/off, changing the event handler structure, and changing the data collection methodology.
- *Sensor query and data dissemination protocol (SQDDP):* Some of the WSN systems require sending dissemination commands from the gateway node to the rest of the nodes

to change of the network operating parameters, such as the data-querying rate. This thesis framework relied on analyzing the power consumption of the network behaviour using the simulator. In addition, the simulation results are integrated with the UML model using the parsing tool. Avrora simulator, which was leveraged in this thesis, supports the packet injection feature. The feature simulates any dissemination commands sent by the gateway. The power consumption of any node activity is analyzed by the simulator and creates an entry in the node log files (see Section 1.4.2). All the log files are parsed using the parsing tool that was implemented as part of the thesis framework.

7.7 Summary

This chapter presented SensIV system as a case study to validate the approach that was developed in this thesis. Explained in this chapter was the structure of SensIV system and the system requirements in order to demonstrate how the developed UML patterns and the power consumption analysis can assist the designer to represent the design at higher abstraction layers and improve the design for the power consumption. SensIV design is represented using five classes (i.e. one class represents the software design and four classes represent the OS components). The UML executed diagrams ware used to validate the procedural requirements of SensIV system, such as the sensing timing and the signalled sensors. In addition, the power consumption of the sensor node was analyzed using the executed and the instrumented state-charts. The sensor node analysis was demonstrated through testing 3 scenarios of controlling the sensors and ADC in order to select the best energy efficient scenario. The power consumption of the sensor network was analyzed through generating the code as well as the simulator configuration files. The simulation results were parsed to feedback the simulation results to the UML model so that the designer can improve the power consumption of the design. The parsing

results enable the designer to improve the network power consumption through modifying the network logic, modifying the event handler's structure, switching the data collection methodology, and changing the network configuration.

8 Summary and Conclusion

8.1 Summary

Chapter 1 introduced the motivation of the thesis work by explaining the challenges of the design process of WSNs. In addition, the chapter introduced the scope of research, the thesis statement, and a brief introduction about the tools that were leveraged in this thesis work. Chapter 2 explained the WSN design background by explaining the WSN structure, WSN power consumption, and WSN modeling. The existing modeling techniques for WSN were surveyed in Chapter 3. Chapter 3 classified the modeling techniques based on their representation for the sensor node at the modeling layer, sensor network at the modeling layer, and the analysis tools that are associated with each modeling technique.

This thesis has presented a group of UML design patterns that enable the designer to represent the WSN at higher abstraction layers and to analyze the power consumption of sensor node components (Chapter 4). In addition, this thesis presented a code generator that generates the required code and the required configuration for WSN simulation tool so that the designer can analyze the power consumption of the sensor network communication process (Chapter 5). The simulation results were integrated with the UML design through parsing the simulator log file so that the designer can improve the design at the modeling layer (Chapter 6). The thesis methodology was evaluated through applying the UML design patterns and power consumption methodologies to a typical WSN collector system called SensIV (Chapter 7).

8.2 Thesis Contribution

This thesis developed a framework and methodology for model level analysis for WSNs. The framework allows the designer to represent the design at the modeling layer and enables the designer to early detect the design errors, validate the design against the system

requirements, and improve the power consumption of the design. The thesis leverages the simulation tools to analyze the networking power consumption and integrates the simulation results with the design representation at the modeling layer (i.e. UML models). The developed framework contains the following developed tools:

1. This thesis presented a UML modeling approach to capture the WSN design at higher abstraction layers other than the coding level. Three UML design patterns were defined which enabled the designer to capture the design structure and the design behaviour by capturing the design OS support components, the component's associations, and the event/events handlers of the design. The UML design pattern enable the designer to validate the operational requirements occurred by using the sequence diagram comparison feature that is supported by IBM rational rhapsody. The UML model execution feature enables the designer to analyze the power consumption of the sensor node components (i.e. ADC, Sensors, CPU, and LEDs). The UML diagrams are instrumented using the power consumption values for each hardware component and are instrumented by the timing values of each component. The timing and power consumption instrumentation are essential for the UML model execution to validate the operational requirement and to analyze the power consumption.
2. This thesis presented a code generator developed to parses the model information and generate the proper code for the simulator and the simulator configuration. The code generator instruments the code through the debugging statements for two purposes: verifying that the code semantics match the design semantics and integrating the simulation results with the model.

3. This thesis presented a methodology to integrate the UML design and Avrora simulation so that designer can estimate the power consumed by the sensor network at the modeling layer. The integration was achieved through using the debugging statements that were generated by the code generator and through using the parsing tool that was implemented as part of the thesis contribution. The debugging statements are displayed at the start and the end of the CPU cycles of the event handlers and the call operations. The implemented parsing tool used the cycle's information and calculated the power consumption of each node through searching the power consumption log files entries that were identified by CPU cycles as well.

8.3 Future Work

This thesis presented an approach to capture the design at higher abstraction layers other than the coding level. In addition, this approach is capable of analyzing the design power consumption and provides details about the power consumption of each hardware component by instrumenting the UML diagrams and by integrating WSN analysis results with the UML models.

As future work, the developed approach can be extended to analyze the non-functional requirements, such as the system latency, since some WSN systems require hard-deadlines in the performance. The system latency requires calculating the worst case execution time (WCET) factor of the OS support components of the design and the delay periods to deliver the package from the source node to the destination node. The UML model analysis approach developed in this thesis can be extended to analyze the software and hardware latency by instrumenting the state-charts with the WECT factor. Additionally, the integration of UML models and the simulation tools can be used to analyze the network delay to deliver the packages.

The current approach can be extended to develop WSN safety systems. The WSN safety systems are considered Wireless Sensor Actor Network systems where the system gathers the information and reacts in response to the gathered information to avoid risks. Based on the safety protocols, such as ISO 262626 protocols, the designer defines the system hazards risks and the safety requirements. The safety requirements normally involves the response time of the system, the predefined actions, or the system states to avoid risks.

Based on the safety protocols, the validation procedure of the safety requirements occurs at the early stages of the design process before the actual implementation of the system. In addition, the design process requires back-to-back code verification process, which is verifying that the code semantics match the design semantics. This thesis approach can validate the safety requirements at the modeling level using the UML diagram instrumentation and the sequence diagram comparison feature. In addition, this approach is capable of developing the back-to-back verification by instrumenting the generated code with debugging statements and compared it to the system sequence diagrams.

As an improvement to the developed approach, the code generator can be extended to support multiple platforms other than TinyOS, which is study case used in this thesis. The code generator can produce the code for event-based operating systems. However, the code generator database needs to be updated with the programming language syntax.

In addition, the current approach relies on using three domains to complete the development cycle: the IBM rational rhapsody to create the model and generate the XML representation for the model; the code generator and parsing tool that generate the nesC code and parses the simulator information respectively; and the Avrora simulator that simulates the network behaviour and calculates the power consumption. The approach used in this thesis relied

on exporting files from each domain and injecting the files to the next domain. Incorporating the code generator and the simulator into one tool will ease using this approach to complete the analysis. Therefore, the designer must inject the XML design representation to the new tool and the new tool will be responsible to generate the code, run the simulation, and parse the simulation results.

9 Reference

1 R. Liscano, J.K. Jacoub, A. Dersingh, J. Zheng, M. Helme, C. Elliott, A. Najafizadeh. Network Performance of a Wireless Sensor Network for Temperature Monitoring in Vineyards. In *8th ACM PE-WASUN* (Miami, USA 2011).

2 R. Beckwith, D. Teibel, P. and Bowen. Unwired wine: sensor networks in vineyards. *IEEE Sensors* (2004), 561-564.

3 J. McCulloch, P. Guru McCarthy, W. Peng, D. Hugo, A. Terhorst. A. Wireless sensor network deployment for water use efficiency in irrigation. In *In Workshop Real-World Wireless Sensor Networks (REALWSN)* (2008).

4 A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Wireless sensor networks for habitat monitoring. In 1st ACM international Workshop on Wireless Sensor Networks and Applications. WSNA* (2002), 88-97.

5 B. Majone, F. Viani, E. Filippi, A. Bellin, A. Massa, G. Toller, F. Robol, M. Salucci. Wireless Sensor Network Deployment for Monitoring Soil Moisture Dynamics at the Field Scale. *Procedia Environmental Sciences*, 19 (2013), 426-435.

6 I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38, 4, 393-422.

7 F. LOSILLA, C. VICENTE-CHICOTE, B. LVAREZ, A. IBORRA, P. SÁNCHEZ. Wireless sensor network application development; An architecture-centric mde approach. *In Software Architecture*, 4758 (2007).

8 O. Sharma, J. Lewis, A. Miller, A. Dearle, D. Balasubramaniam, R. Morrison, and J. Svente.

Towards verifying correctness of wireless sensor network applications using Insense and Spin. *Lecture Notes in Computer Science,* 223–240.

9 IBM. *Rational Rhapsody User Guide*. IBM, 2000-2009.

10 B.L. Titzer, D.K. Lee, J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks (IPSN)* (2005).

11 R. Pesch, D. Alberola. AvroraZ: extending Avrora with an IEEE 802.15.4 compliant radio chip model. In *nd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks* (2008).

12 A. Dearle, D. Balasubramaniam, J. Lewis, and R. Morrison. A component-based model and language for wireless sensor netowrk applications. In *In Proc. of 32nd Annual IEEE Int.Conf. on Computer Software and Applications (COMPSAC)* (Aug. 2008.), pages 1303–1308.

13 ITU-T. *Specification and description language (SDL), z. 100(11/99) edition, 1999*.

14 D. Dietterle, J. Ryman, K. Dombrowski, and R. Kraemer. Mapping of high-level SDL models to efficient implementations for TinyOS. In *In Proc. of Euromicro Symp. on Digital System Design (DSD 2004)* (Sept 2004), pages 402-406.

15 A. Dunkels, B. Gronvall, T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. (2004), 455- 462.

16 M. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, S. Olivieri. A framework for modeling, simulation and automatic code generation of sensor network application. In *In Proc. of 5th IEEE Comm. Soc. Conf. on Sensor, Mesh and Ad-Hoc* (2008).

17 S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A.

Torgerson, and R. Han. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications,* (2005), 563–579.

18 L. Necchi, A. Bonivento, L. Lavagno, A. Sangiovanni, and L. Vanzago. E2rina: an energy efficient and reliable in-network aggregation for clustered wireless sensor netowrks. In *In Wireless Communications and Networking Conference* (2007), 3364 –3369.

19 V. Oleshchuk. Ad-hoc sensor networks: modeling, specification and verfication. In *In Proc. of 2nd IEEE Int. Work on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications* (Sept. 2003), pages 76-79.

20 M. Botts, A. Robin. *OpenGIS sensor model language (SensorML) implmentation specification Technical report, OGC*. 2007.

21 M. Vasilevski, N. Beilleau, H. Aboushady, and F. Pecheux. Efficient and refined modeling of wireless sensor network nodes using SystemC-AMS. In *In Conf. on Ph.D. Research in Microelectronics and Electronics (PRIME 2008)* (Apr. 2008), 81-84.

22 M. Diaz, D. Garrido, L. Llopis, B. Rubio, and J. Troya. A component framework for wireless sensor and actor networks. In *In Proc. of IEEE Conf. on Emerging Technologies and Factory Automation (ETFA '06)* (Sept. 2006), 300–307.

23 J. Barbaran, M. Diaz, I. Esteve, D. Garrido, L. Llopis, B. Rubio, and J. Troya. Tc-wsan. A tuple channel based coordination model for wireless sensor and actor networks. In *In Computers and Communications ISCC* (2007), 173-178.

24 A. Demaille, S. Peyronnet, and B. Sigoure. Modeling of sensor networks using XRM. In *In Proc. of 2nd Int. Symp. on Leveraging Applications of Formal Methods, Verification*

and Validation (Nov. 2006.), 217-276.

25 C. Reichmann, M. Kiihl, P. Graf, K.D. Müller-Glaser. GeneralStore - a CASE-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. *Engineering of Computer-Based Systems* (2004).

26 C. Sommer, I. Dietrich, F.lko Dressler, W. Dulz, and R. German. A tool Chain for UML-Based Modeling and Simulation of VANET Scenarios. In *9th ACM International Symposium on Mobil Ad Hoc Networking and Computing* (2008).

27 M. Woodside, D. C. Petriu, J. Merseguer, D, B. Petrie, M. Alhaj, ", DOI: 10.1007/s10270-013-0385-x, published online.. Transformation challenges: from software models to performance models".

28 S. Bernardi, J. Merseguer, D.C. Petriu. Dependability modeling and analysis of software systems specified with UML. *ACM Computing Surveys (CSUR* (2012).

29 F. Losilla, C. Vicente-Chicote, B. lvarez,A. Iborra, P. Sánchez. Wireless sensor network application development; An architecture-centric mde approach. *In Software Architecture*, 4758 (2007), 197-194.

30 M. Robin, A. Botts. *OpenGIS sensor model language (SensorML) implementation specification.* Jul. 2007.

31 O. Gnawal, R. Fonseca, K. Jamieson, D. Moss, P. Levis. Collection Tree Protocol. In *7th ACM Conference on Embedded Networked Sensor Systems* (2009).

32 P. Levis. *TinyOS Programming. Cambridge University Press*. 2009.

33 T. Watteyne, K. Pister, D. Barthel, M. Dohler, I. Auge-Blum. Implementation of Gradient Routing in Wireless Sensor Networks. In *Global Telecommunications Conference*

(2009).

34 E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

35 J. Zheng, C. Elliott, A. Dersingh, R. Liscano, M. Eklund. Design of a Wireless Sensor Network from an Energy Management Perspective. In *Communication Networks and Services Research Conference (CNSR)* (2010), 80-86.

36 Rhapsody, IBM Rational. *IBM Rational Rhapsody Reference Workflow Guide for ISO 26262*. 2013.

37 J. ZHENG, C. ELLIOTT, A. DERSINGH, R. LISCANO, M. EKLUND. Design of a Wireless Sensor Network from an Energy Management Perspective. In *Communication Networks and Services Research Conference (CNSR)* (2010).

38 W. Li, M. Bandai, T. Watanabe. radeoffs among Delay, Energy and Accuracy of Partial Data Aggregation in Wireless Sensor Networks. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on* (2010), 917-924.

39 O. Landsiedel, K. Wehrle, S. Gotz. Accurate Prediction of Power Consumption in Sensor Networks. In *Workshop on Embedded Systems* (2005).

40 J. Yick, B. Mukherjee, D. Ghosal. Wireless sensor network survey. *Computer Networks*, 52 (August 2008).

41 J.M. Kahn, R.H. Katz, K.S.J. Pister. Next century challenges: mobile networking for smart. In *Proceedings of the ACM MobiCom'99,* (1999).

42 Z. Xiong, Z. Song, A. Scalera, E. Ferrera, F. Sottile, P. Brizzi, R. Tomasi, M. A Spirito. Hybrid WSN and RFID indoor positioning and tracking system. *EURASIP on Embedded*

Systems (2013).

43 *Tinyos 2 tep 123. The collection tree protocol.* http://www.tinyos.net/tinyos-2.x/doc/txt/tep123.txt.

44 R. CHEN, M. SGROI, L. LAVAGNO, G. MARTIN, A. SANGIOVANNI-VINCENTELLI, J. RABAEY. Embedded System Design using UML and Platforms. In , ed., *System Specification & Design Languages*. Springer US.

45 Crossbow. *Iris Datasheet*. Crossbow.

46 A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *In Proc. of 29th Annual IEEE Int. Conf. on Local Computer Networks* (Nov. 2004.), 455–462.

47 Lu., B. Nickerson and J. A language for wireless sensor webs. In *In Communication Networks and Services Research Proceedings. Second Annual Conference* (2004), 293-300.

48 E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1990.

49 J. K. Jacoub, R. Liscano, R., J.S. Bradbury. Assessment of Software Modeling Techniques for Wireless Sensor Networks: A Survey. *Sensors & Transducers Journal*, 14-2 (2012), 18-46.

50 A. Demaille, A. Peyronnet, S. Sigoure. Modeling of sensor networks using XRM. In *In Proc. of 2nd Int. Symp. on Leveraging Applications of Formal Methods, Verification and validation* (2006).

51 P. Levis, P., D. Gay *TinyOS Programming*. Combridge University Press, 2009.

52 J.K. Jacoub, R. Liscano, J.S. Bradbury, UML ModelLing of Design Patterns for Wireless Sensor Networks. In *sensornet2013* (2013).

53 R. Liscano, J.K. Jacoub, A. Dersingh, J. Zheng, M. Helme, C. Elliott, A. Najafizadeh. Network Performance of a Wireless Sensor Network for Temperature Monitoring in Vineyards. In *8th ACM PE-WASUN* (Miami, USA 2011).

54 M. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, S. Olovieri, A framework for modeling, simulation and automatic code generation of sensor network application. In *In Proc. of 5th IEEE Comm. Soc. Conf. on Sensor, Mesh and Ad-Hoc* (2008).

55 L. Meijun, S. Zhaohua. Research on Quality of Service in Wireless Sensor Networks. In *Information Engineering and Computer Science (ICIECS)* (2010), 1-4.

56 J.K. Jacoub, R. Liscano, J.S. Bradbury. Assessment of Software Modeling Techniques for Wireless Sensor Networks: A Survey. *Sensors & Transducers Journal*, 14-2 (2012), 18-46.

57 E. Perla, A. Catháin, R.C. Carbajo, M. Huggard, M.C. Goldrick. PowerTOSSIM z: Realistic Energy Modelling for Wireless Sensor Network Environments. In *Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks* (2008).

58 J.K. Jacoub, R. Liscano, J.S. Bradbury, J. Fisher. UML Modeling And Analysis of Power Consumption For Wireless Sensor Network. In *sensornet* (2013).

10 Appendices

Appendix I

The appendix shows the generated code for SensIV system.

```
#include "AvroraPrint.h"
module SensIV{
uses interface Boot;
uses interface Leds as Leds;
uses interface GeneralIO as myPW1;
uses interface SplitControl as myRadio;
uses interface StdControl as my;
uses interface Timer<TMilli> as myTimer;
uses interface Read<uint16_t> as mySensor0;
uses interface Read<uint16_t> as myBattery;
uses interface GeneralIO as myPW7;
uses interface Timer<TMilli> as myTimer2;
uses interface StdControl as myRouting;
uses interface Read<uint16_t> as mySensor1;
uses interface Read<uint16_t> as mySensor2;
uses interface Read<uint16_t> as mySensor3;
uses interface Send as mySend;
uses interface Receive as myReceive;
}
implementation  {
bool sendbusy=FALSE; ;
uint16_t reading[];;
uint16_t voltage=380; ;
message_t sendbuf;

event void Boot.booted() {

printStr("|BootedHandler| St");
printStr("|Leds_S7|");
call Leds.led0Off();
call Leds.led1Off();
call Leds.led2Off();
printStr("|TurnoffADC|");
call myPW1.set();
printStr("|StartRadio|");
call myRadio.start();
printStr("|StartRouting|");
call my.start();
printStr("|BootedHandler| En");
```

```
event void myTimer2.fired() {
printStr("|WarmUpTimerHandler| St");
printStr("|TriggerSensor0");
call mySensor0.read();
printStr("|WarmUpTimerHandler| En");
}

event void myTimer.fired() {
printStr("|SamplerTimerHandler| St");
printStr("|Leds_S1|");
call Leds.led0Off();
call Leds.led1Off();
call Leds.led2On();
printStr("|ReadVoltage|");
call myBattery.read();
printStr("|TurnOnADC|");
call myPW7.makeOutput();
call myPW7.clr();
call myPW1.makeOutput();
call myPW1.set();
printStr("|StartTimerS2|");
call myTimer2.startPeriodic(1000);
printStr("|SamplerTimerHandler| En");
}
event void myBattery.readDone(error_t result, uint16_t data) {
printStr("|Vref_readDone Handler| St");
printStr("|Leds_S4|");
call Leds.led0On();
call Leds.led1Off();
call Leds.led2On();
printStr("|Vref_readDone Handler| En");
}
event void myRadio.stopDone(error_t error) {
printStr("|StopRadio| St");
call Leds.led0On();
call Leds.led1On();
call Leds.led2Off();
call myRadio.start();
call myRouting.start();
printStr("|StopRadio| En");
}
event void mySensor1.readDone(error_t result, uint16_t data) {
printStr("|Sensor1readDoneHanlder| St");
printStr("|Leds_S5|");
call Leds.led1On();
call Leds.led0On();
call Leds.led2On();
printStr("|TriggerSensor2");
call mySensor2.read();
printStr("|Sensor1readDoneHanlder| En");
}
```

```
event void mySensor2.readDone(error_t result, uint16_t data) {
printStr("|Sensor2readDoneHandler| St");
printStr("|Leds_S5|");
call Leds.led1On();
call Leds.led0On();
call Leds.led2On();

printStr("|TriggerSensor2");
call mySensor3.read();
printStr("|Sensor2readDoneHandler| En");
}
event void mySensor3.readDone(error_t result, uint16_t data) {
printStr("|Sensor3readDoneHandler| St");
printStr("|Leds_S5|");
call Leds.led1On();
call Leds.led0On();
call Leds.led2On();
printStr("|TurnoffADC|");
call myPW1.set();
printStr("|StartTimerS1|");
call myTimer.startPeriodic(5000);
printStr("|SendData|");
call mySend.send(&sendbuf, 56);
printStr("|Sensor3readDoneHandler| En");
}
event message_t* myReceive.receive(message_t*msg, void* payload, uint8_t len) {
printStr("|ReceiveHanlder| St");
printStr("|Leds_S3|");
call Leds.led0Off();
call Leds.led1On();
call Leds.led2On();
returnmsg;
printStr("|ReceiveHandler| En");
}
}
```

Appendix II

This appendix shows the energy log file for one of the sensor node. The log file shows the hardware components, the CPU cycle for each power consumption entry, and the power consumed value for each hardware component during each processor cycle.

```
cycle CPU Yellow Green Red Radio SensorBoard flash total
0 0.0075667 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0082887
8011181 0.0075667 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0082887
8011182 0.0075667 0.0 0.0 0.0022 2.0E-5 7.0E-4 2.0E-6 0.010488699999999998
8011183 0.0075667 0.0 0.0 0.0022 2.0E-5 7.0E-4 2.0E-6 0.010488699999999998
8011184 0.0075667 0.0 0.0022 0.0022 2.0E-5 7.0E-4 2.0E-6 0.012688699999999999
8011185 0.0075667 0.0 0.0022 0.0022 2.0E-5 7.0E-4 2.0E-6 0.012688699999999999
8011186 0.0075667 0.0022 0.0022 0.0022 2.0E-5 7.0E-4 2.0E-6 0.0148887
8011187 0.0075667 0.0022 0.0022 0.0022 2.0E-5 7.0E-4 2.0E-6 0.0148887
8011188 0.0075667 0.0022 0.0022 0.0 2.0E-5 7.0E-4 2.0E-6 0.012688699999999999
8011189 0.0075667 0.0022 0.0022 0.0 2.0E-5 7.0E-4 2.0E-6 0.012688699999999999
8011190 0.0075667 0.0022 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.010488699999999998
8011191 0.0075667 0.0022 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.010488699999999998
8011192 0.0075667 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0082887
8013452 0.0075667 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0082887
8013453 0.0033433 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0040653
8017864 0.0033433 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0040653
8017865 0.0075667 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0082887
8026224 0.0075667 0.0 0.0 0.0 2.0E-5 7.0E-4 2.0E-6 0.0082887
8026225 0.0075667 0.0 0.0 0.0 4.26E-4 7.0E-4 2.0E-6 0.0086947
8027496 0.0075667 0.0 0.0 0.0 4.26E-4 7.0E-4 2.0E-6 0.0086947
8027497 0.0075667 0.0 0.0 0.0 0.0188 7.0E-4 2.0E-6 0.027068699999999998
8029314 0.0075667 0.0 0.0 0.0 0.0188 7.0E-4 2.0E-6 0.027068699999999998
8029315 0.0033433 0.0 0.0 0.0 0.0188 7.0E-4 2.0E-6 0.0228453
9661274 0.0033433 0.0 0.0 0.0 0.0188 7.0E-4 2.0E-6 0.0228453
9661275 0.0075667 0.0 0.0 0.0 0.0188 7.0E-4 2.0E-6 0.027068699999999998
9661599 0.0075667 0.0 0.0 0.0 0.0188 7.0E-4 2.0E-6 0.027068699999999998
9661600 0.0033433 0.0 0.0 0.0 0.0188 7.0E-4 2.0E-6 0.0228453
……………….
```

Appendix III

This appendix shows the trace file for the nodes, the CPU cycle, and the output of the debugging statements.

```
Loading Node.elf...[OK: 0.125 seconds]
=======================================================
        Node        Time   Event
        ----------------------------------------------------------------------------
          0     8012848  |Booted| St
          0     8012992  |RoutingStart| St
          0     8013392  |Booted| En
          0     8027833  |RoutingStart| En
          0     8028005  |StartDoneHandler| St
          0     8028149  |StartTimerS1| St
          0     8028431  |StartTimerS1| St
          0     8028728  |StartDoneHandler| En
          8     8012848  |Booted| St
          8     8012992  |RoutingStart| St
          8     8013392  |Booted| En
          8     8027833  |RoutingStart| En
          8     8028005  |StartDoneHandler| St
          8     8028149  |StartTimerS1| St
          8     8028431  |StartTimerS1| St
          8     8028728  |StartDoneHandler| En
          5     8012848  |Booted| St
          5     8012992  |RoutingStart| St
          5     8013392  |Booted| En
          5     8027833  |RoutingStart| En
          5     8028005  |StartDoneHandler| St
          5     8028149  |StartTimerS1| St
          5     8028431  |StartTimerS1| St
          5     8028728  |StartDoneHandler| En
          4     8012848  |Booted| St
          4     8012992  |RoutingStart| St
          4     8013392  |Booted| En
          4     8027833  |RoutingStart| En
          4     8028005  |StartDoneHandler| St
          4     8028149  |StartTimerS1| St
          4     8028431  |StartTimerS1| St
          4     8028728  |StartDoneHandler| En
          9     8012848  |Booted| St
          9     8012992  |RoutingStart| St
          9     8013392  |Booted| En
          9     8027833  |RoutingStart| En
          9     8028005  |StartDoneHandler| St
          9     8028149  |StartTimerS1| St
          9     8028431  |StartTimerS1| St
          9     8028728  |StartDoneHandler| En
          3     8012848  |Booted| St
        …………..
```