Model-driven development of sensor network applications with optimization of non-functional constraints

Dissertation for the acquisition of the academic degree

Doktor der Naturwissenschaften (Dr. rer. nat.)

Submitted to the
Faculty of Electrical Engineering and Computer Science
of the University of Kassel, Germany
by M.Sc. Nguyen Xuan Thang

Kassel, June 2015

Advisors:
Prof. Dr. Kurt Geihs
Prof. Dr. Torben Weis

Additional Doctoral Committee Members:
Prof. Dr. Michael Schwarz
Prof. Dr. Albert Zündorf

Date of Defense:
Thursday, 08 October 2015
# Contents

Contents................................................................................................................................................ i

List of Figures.................................................................................................................................... v

List of Tables................................................................................................................................... vii

List of Listing.................................................................................................................................... ix

List of Abbreviations..................................................................................................................... xi

Abstract ........................................................................................................................................... xiii

Zusammenfassung ........................................................................................................................ xv

Acknowledgements .................................................................................................................... xvii

Part I  Foundation ........................................................................................................................... 1

1 Introduction...................................................................................................................................... 1

1.1 Motivation ................................................................................................................................................... 1

1.2 Problem statement .................................................................................................................................. 3

1.2.1 Problem analysis ............................................................................................................................. 3

1.2.2 Research objective ...................................................................................................................... 11

1.3 Research method ................................................................................................................................ 12

1.4 Solution approach ................................................................................................................................ . 13

1.5 Structure of the thesis ......................................................................................................................... 17

2 Wireless sensor network ........................................................................................................... 19

2.1 WSN architecture and hardware technology ............................................................................. 19

2.1.1 Network architecture ................................................................................................................. 19

2.1.2 Hardware technology ................................................................................................................. 20

2.2 Communication protocols and standardizations ..................................................................... 22

2.2.1 Physical layer ................................................................................................................................ . 22

2.2.2 Data-link layer ................................................................................................................................ 24

2.2.3 Network layer ................................................................................................................................ 28

2.2.4 Transport layer ................................................................................................................................ 34

2.2.5 Standardizations .......................................................................................................................... 40

2.3 Operating system .......................................................................................................................... 44

2.3.1 Event-based and thread-based model ......................................................................................... 44

2.3.2 Tiny OS ............................................................................................................................................. 46

2.3.3 Contiki .............................................................................................................................................. 48

2.3.4 Other OSs for motes .................................................................................................................... 51

3 Model-driven software development .................................................................................... 53

3.1 Concept and definition .................................................................................................................... 53
Part II Solution .............................................................................................................................. 73

5 Programming model .................................................................................................................................. 75

5.1 Programming approaches for WSNs ........................................................................................................ 75

5.1.1 Programming languages .................................................................................................................... 75

5.1.2 Programming models .......................................................................................................................... 76

5.2 Node-centric and macro-programming approaches .................................................................................. 76

5.2.1 Node-centric model ............................................................................................................................ 76

5.2.2 Macro-programming model ............................................................................................................... 77

5.3 Rule-based programming model ........................................................................................................... 77

5.3.1 Node behaviors .................................................................................................................................. 78

5.3.2 Message structure ............................................................................................................................... 79

5.3.3 Rule structure ................................................................................................................................... 79

6 Software architecture ................................................................................................................................ 83

6.1 Architecture design .................................................................................................................................. 83

6.1.1 Implementation of the Hardware layer ............................................................................................... 86

6.1.2 Implementation of the Operating System layer .................................................................................. 86

6.1.3 Implementation of the Operating System Abstraction layer ............................................................. 88

6.1.4 Implementation of the Application layer ............................................................................................ 90

6.2 Architecture formalization ...................................................................................................................... 90

6.3 Hardware layer ...................................................................................................................................... 92

6.3.1 Hardware component description ...................................................................................................... 93

6.3.2 Hardware layer interface .................................................................................................................... 94

6.3.3 Hardware layer instance .................................................................................................................... 95

6.4 Operating system layer ............................................................................................................................ 96

6.4.1 Operating system layer interface ....................................................................................................... 96
Part III  Evaluation and Conclusion................................................................................................. 157

11  Framework evaluation.................................................................................................................. 159
   11.1  The ease of WSN programming .............................................................................................. 159
       11.1.1  User study setup .................................................................................................................. 159
       11.1.2  User study result .................................................................................................................. 161
   11.2  Portability evaluation .............................................................................................................. 163
       11.2.1  Mote platforms and benchmark applications .................................................................. 163
       11.2.2  Portability evaluation result .............................................................................................. 164
   11.3  Framework overhead evaluation ............................................................................................. 166
       11.3.1  Overhead evaluation metrics .............................................................................................. 166
       11.3.2  Overhead evaluation result .............................................................................................. 168

12  Design and optimize a real-world application ........................................................................ 175
   12.1  TempSense application ............................................................................................................ 175
   12.2  Optimizing TempSense with SSDOE ...................................................................................... 178
   12.3  Optimizing TempSense with MOEA ....................................................................................... 181

13  Discussion and Conclusion ...................................................................................................... 183
   13.1  Requirement revisited ............................................................................................................. 183
   13.2  Contribution ............................................................................................................................. 185
   13.3  Future work ............................................................................................................................. 186

Part IV  Appendices ......................................................................................................................... 189

Appendix A  WSN technologies ..................................................................................................... 191
   A1  Mote hardware platform evolution ......................................................................................... 191
   A2  Physical layer design requirements ......................................................................................... 192
   A3  MAC protocol analysis ............................................................................................................. 192
   A4  Routing protocol analysis ........................................................................................................ 193
   A5  Transport protocol analysis ...................................................................................................... 193
   A6  Mote's operating system analysis ............................................................................................ 194

Appendix B  Software Architecture ................................................................................................ 196
   B1  Hardware description ................................................................................................................ 196
   B2  Hardware Platform Specific Model of Z1 mote ...................................................................... 198
   B3  OS Platform Specific Model for TinyOS 2.x ......................................................................... 199
   B4  OS Platform Specific Model for Contiki 2.x ......................................................................... 200
   B5  An example of the application.xml file .................................................................................... 201

Appendix C  Bibliography .......................................................................................................... 203

Erklärung ........................................................................................................................................... 215
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Mote platforms evolution</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Software architecture for sensor nodes</td>
<td>9</td>
</tr>
<tr>
<td>1.3</td>
<td>Software architecture for sensor nodes with a Middleware</td>
<td>10</td>
</tr>
<tr>
<td>1.4</td>
<td>The proposed software architecture of a WSN mote</td>
<td>14</td>
</tr>
<tr>
<td>1.5</td>
<td>An overview of the application development process</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>WSN architecture</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Mote’s hardware architecture</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>The difference between three routing schemes</td>
<td>31</td>
</tr>
<tr>
<td>2.4</td>
<td>Comparison of stack usage in event-based and thread-based model</td>
<td>45</td>
</tr>
<tr>
<td>2.5</td>
<td>Comparison of control flow in model-based and thread-based model</td>
<td>45</td>
</tr>
<tr>
<td>2.6</td>
<td>TinyOS architecture</td>
<td>47</td>
</tr>
<tr>
<td>2.7</td>
<td>Contiki OS architecture</td>
<td>49</td>
</tr>
<tr>
<td>2.8</td>
<td>Rime protocol stack</td>
<td>50</td>
</tr>
<tr>
<td>3.1</td>
<td>A sample of model-to-model transformation</td>
<td>54</td>
</tr>
<tr>
<td>3.2</td>
<td>Current approaches to deal with NFR in a MDD framework</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>Classification of programming approaches for WSNs</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Working behaviors of a node in WSN</td>
<td>78</td>
</tr>
<tr>
<td>6.1</td>
<td>The proposed software architecture for WSN mote</td>
<td>84</td>
</tr>
<tr>
<td>6.2</td>
<td>Component diagram of the mote architecture</td>
<td>85</td>
</tr>
<tr>
<td>6.3</td>
<td>Implementation diagram of the Hardware Layer</td>
<td>87</td>
</tr>
<tr>
<td>6.4</td>
<td>Implementation diagram of the Operating System Layer</td>
<td>88</td>
</tr>
<tr>
<td>6.5</td>
<td>Implementation diagram of the Operating System Abstraction Layer</td>
<td>89</td>
</tr>
<tr>
<td>6.6</td>
<td>The implementation diagram of the Application Layer</td>
<td>90</td>
</tr>
<tr>
<td>7.1</td>
<td>WSN application classification</td>
<td>106</td>
</tr>
<tr>
<td>7.2</td>
<td>Proposed DSL metamodel</td>
<td>110</td>
</tr>
<tr>
<td>7.3</td>
<td>Graphical DSL framework development using GMF</td>
<td>113</td>
</tr>
<tr>
<td>7.4</td>
<td>An example of PIM</td>
<td>115</td>
</tr>
<tr>
<td>8.1</td>
<td>Overview of the transformation process</td>
<td>117</td>
</tr>
<tr>
<td>8.2</td>
<td>Instance component diagram of the Operating System Abstraction Layer</td>
<td>118</td>
</tr>
<tr>
<td>9.1</td>
<td>An example of COOJA GUI</td>
<td>131</td>
</tr>
<tr>
<td>9.2</td>
<td>The operation of X-MAC protocol</td>
<td>135</td>
</tr>
<tr>
<td>9.3</td>
<td>The Timeline of X-MAC simulation in COOJA</td>
<td>136</td>
</tr>
<tr>
<td>10.1</td>
<td>An example of Pareto frontier</td>
<td>141</td>
</tr>
<tr>
<td>10.2</td>
<td>Two different experiment scenarios</td>
<td>143</td>
</tr>
<tr>
<td>10.3</td>
<td>The impacts of rebroadcast probability on applications non-functional properties</td>
<td>145</td>
</tr>
<tr>
<td>10.4</td>
<td>The impacts of sleep time on the application non-functional properties</td>
<td>146</td>
</tr>
<tr>
<td>10.5</td>
<td>The optimization component design</td>
<td>155</td>
</tr>
<tr>
<td>11.1</td>
<td>Groups' members and their programming background</td>
<td>160</td>
</tr>
<tr>
<td>11.2</td>
<td>Detail of the user study result</td>
<td>162</td>
</tr>
</tbody>
</table>
Figure 11.3 The benchmark applications evaluation process ................................................................. 164
Figure 11.4 Summary of the framework portability evaluation result ....................................................... 166
Figure 11.5 Comparison of memory footprints and image sizes between Read Sensor applications ........................................................... 169
Figure 11.6 Comparison of memory footprints and image sizes between Read Sensor and Blink applications .................................................................................. 169
Figure 11.7 Comparison of memory footprints and image sizes between Broadcasting applications ........................................................................................................... 170
Figure 11.8 Comparison of memory footprints and image sizes between Periodic Broadcasting applications.................................................................................................................. 171
Figure 11.9 Comparison of memory footprints and image sizes between Multi-hop Sending applications ................................................................................................................................. 171
Figure 11.10 Memory footprint overhead introduced by the framework on benchmark applications for TinyOS ................................................................................................................. 172
Figure 11.11 Memory footprint overhead introduced by the framework on benchmark applications for Contiki ....................................................................................................................... 172
Figure 11.12 Binary image size overhead introduced by the framework on benchmark applications for TinyOS ................................................................................................................... 173
Figure 11.13 Binary image size overhead introduced by the framework on benchmark applications for Contiki .................................................................................................................. 173
Figure 12.1 A PIM of TempSense application ............................................................................................. 176
Figure 12.2 The objective space with one million points when optimizing TempSense .................................... 179
Figure 12.3 The Pareto frontier obtained when optimizing TempSense application using SSDOE ......................... 180
Figure 12.4 The Pareto frontier obtained when optimizing TempSense application using SPEA2 ....................... 181
Figure 12.5 Comparison between two approaches SSDOE and SPEA2 ......................................................... 182
Figure B.1 Hardware Platform Specific Model for Z1 mote ........................................................................ 198
Figure B.2 OS Platform Specific Model for TinyOS 2.x ........................................................................... 199
Figure B.3 OS Platform Specific Model for Contiki 2.x ........................................................................... 200
List of Tables

Table 1.1 An example of mote platforms diversity ............................................................................................................................................................................................... 4
Table 1.2 Blink application for different operating systems ................................................................................................................................................................................. 6
Table 2.1 Main characteristics of event-based and thread-based model .................................................................................................................................................................................. 46
Table 3.1 Relationships between NFRs attributes [213] .................................................................................................................................................................................. 57
Table 4.1 Middleware approaches in WSN .......................................................................................................................................................................................... 62
Table 5.1 Examples of SELECT clause ........................................................................................................................................................................................ 80
Table 5.2 List of possible node attribute, application parameters, and pre-defined functions ........................................................................................................................................................................... 81
Table 5.3 List of supported in-network processing functions ........................................................................................................................................................................... 81
Table 5.4 List of available rule actions ........................................................................................................................................................................................ 82
Table 6.1 List of the hardware layer interfaces .......................................................................................................................................................................................... 95
Table 6.2 TinyOS 2.x interface for LED services .................................................................................................................................................................................. 97
Table 6.3 TinyOS 2.x interface for Clock & Power saving services ............................................................................................................................................................................. 98
Table 6.4 TinyOS 2.x interface for Sensor and Actuator services .............................................................................................................................................................................. 98
Table 6.5 TinyOS 2.x interface for Scheduling service .................................................................................................................................................................................... 98
Table 6.6 TinyOS 2.x interface for Storage services .................................................................................................................................................................................... 99
Table 6.7 TinyOS 2.x interface for Network services ................................................................................................................................................................................ 100
Table 6.8 Contiki interface for LED services .................................................................................................................................................................................... 101
Table 6.9 Contiki interface for Clock services .................................................................................................................................................................................... 101
Table 6.10 Contiki interface for Sensor and Actuator services ............................................................................................................................................................................. 102
Table 6.11 Contiki interface for Scheduling services .................................................................................................................................................................................... 102
Table 6.12 Contiki interface for Storage services .................................................................................................................................................................................... 103
Table 6.13 Contiki interface for Networking services ................................................................................................................................................................................... 104
Table 7.1 Summarize of WSN non-functional properties ................................................................................................................................................................................. 109
Table 7.2 Graphical notation of the proposed DSL .................................................................................................................................................................................. 115
Table 8.1 List of C function library .......................................................................................................................................................................................... 122
Table 8.2 Main block translation to native code in TinyOS 2.x and Contiki ............................................................................................................................................................................. 124
Table 8.3 Regular expressions and patterns for translating functions ................................................................................................................................................................... 126
Table 9.1 The power consumption of Z1’s components in different states ..................................................................................................................................................................... 132
Table 9.2 List of WSN metrics collected by the plugin during a simulation ........................................................................................................................................................................... 133
Table 10.1 Summary of the user study result ......................................................................................................................................................................................... 161
Table 10.2 Selected mote platforms for portability evaluation ............................................................................................................................................................................. 163
Table 10.3 Benchmark applications for portability evaluation ............................................................................................................................................................................. 164
Table 10.4 Portability evaluation result ........................................................................................................................................................................................ 165
Table 10.5 An example of memory footprint and image size of Read Sensor application ........................................................................................................................................................................... 168
Table 12.1 List of adjustable parameters and non-functional constrains for TempSense application ........................................................................................................................................................................... 177
Table 12.2 Obtained statistical models to predict two non-functional metrics from input parameters in TempSense ........................................................................................................................................................................... 178
Table 12.3 An example set of ten solutions obtained when optimizing TempSense application using SSDoE ........................................................................................................................................................................... 180
Table A.1 Mote hardware platform evolution from 2002 to 2012 ........................................................................................................................................................................... 191
Table A.2 Overview of physical layer design requirements in WSN ......................................................192
Table A.3 Overview of the representative MAC protocols .........................................................................193
Table A.4 Overview of the representative routing protocols ....................................................................193
Table A.5 Overview of the representative transport protocols ...............................................................194
Table A.6 Comparison of mote's operating system.......................................................................................195
List of Listing

Listing 6.1 An algorithm for generating the mapping between Operating System Translator Layer and Operating System Layer ........................................................................................................................ 92
Listing 6.2 A XML file to describe Digital Temperature sensor TMP102 on Z1 Mote ................. 94
Listing 8.1 An example of application configuration file................................................................. 119
Listing 10.1 Algorithm checks whether a vector u is dominated by a vector v.............................. 149
Listing 10.2 Algorithm computes all points in the Pareto frontier..................................................... 150
Listing 10.3 Strength Pareto Evolution Algorithm 2........................................................................ 152
Listing 11.1 Application memory layout returned by msp430-size utility ...................................... 167
Listing B.1 An example of temperature sensor specification.............................................................. 197
Listing B.2 A XML file to describe the example PIM....................................................................... 202
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog-to-Digital Converter</td>
</tr>
<tr>
<td>APL</td>
<td>Application Layer</td>
</tr>
<tr>
<td>CIM</td>
<td>Computation Independent Model</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal–Oxide–Semiconductor</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>EA</td>
<td>Evolutionary Algorithm</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read-Only Memory</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>FR</td>
<td>Functional Requirement</td>
</tr>
<tr>
<td>GEF</td>
<td>Graphical Editing Framework</td>
</tr>
<tr>
<td>GMF</td>
<td>Graphical Modeling Framework</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>HL</td>
<td>Hardware Layer</td>
</tr>
<tr>
<td>HLI</td>
<td>Hardware Layer Interface</td>
</tr>
<tr>
<td>ISM-Band</td>
<td>Industrial, Scientific and Medical Band</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>MOEA</td>
<td>Multi-Objective Evolutionary Algorithm</td>
</tr>
<tr>
<td>MOF</td>
<td>The Meta Object Facility</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-Functional Property</td>
</tr>
<tr>
<td>NFR</td>
<td>Non-Functional Requirement</td>
</tr>
<tr>
<td>OAL</td>
<td>Operating System Abstraction Layer</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSL</td>
<td>Operating System Layer</td>
</tr>
<tr>
<td>OSLI</td>
<td>Operating System Layer Interface</td>
</tr>
<tr>
<td>OSTL</td>
<td>Operating System Translator Layer</td>
</tr>
<tr>
<td>OSTLI</td>
<td>Operating System Translator Layer Interface</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-Only Memory</td>
</tr>
<tr>
<td>SBX</td>
<td>Simulated Binary Crossover</td>
</tr>
<tr>
<td>SPEA</td>
<td>Strength Pareto Evolutionary Algorithm</td>
</tr>
<tr>
<td>SPEA2</td>
<td>Strength Pareto Evolutionary Algorithm 2</td>
</tr>
<tr>
<td>SSDOE</td>
<td>Systematic Statistical Design of Experiments</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Abstract

Wireless sensor networks (WSNs) differ from conventional distributed systems in many aspects. The resource limitation of sensor nodes, the ad-hoc communication and topology of the network, coupled with an unpredictable deployment environment are difficult non-functional constraints that must be carefully taken into account when developing software systems for a WSN. Thus, more research needs to be done on designing, implementing and maintaining software for WSNs. This thesis aims to contribute to research being done in this area by presenting an approach to WSN application development that will improve the reusability, flexibility, and maintainability of the software.

Firstly, we present a programming model and software architecture aimed at describing WSN applications, independently of the underlying operating system and hardware. The proposed architecture is described and realized using the Model-Driven Architecture (MDA) standard in order to achieve satisfactory levels of encapsulation and abstraction when programming sensor nodes. Besides, we study different non-functional constrains of WSN application and propose two approaches to optimize the application to satisfy these constrains.

A real prototype framework was built to demonstrate the developed solutions in the thesis. The framework implemented the programming model and the multi-layered software architecture as components. A graphical interface, code generation components and supporting tools were also included to help developers design, implement, optimize, and test the WSN software.

Finally, we evaluate and critically assess the proposed concepts. Two case studies are provided to support the evaluation. The first case study, a framework evaluation, is designed to assess the ease at which novice and intermediate users can develop correct and power efficient WSN applications, the portability level achieved by developing applications at a high-level of abstraction, and the estimated overhead due to usage of the framework in terms of the footprint and executable code size of the application. In the second case study, we discuss the design, implementation and optimization of a real-world application named TempSense, where a sensor network is used to monitor the temperature within an area.
Zusammenfassung


I would like to take this opportunity to thank those individuals and organizations, without whom this thesis and associated work would not have been possible. First, and foremost, I would like to thank my advisor Prof. Dr. Kurt Geihs for offering me the chance to work in his research group. I greatly appreciate the advice, opinions and insight he has provided throughout my work. I would also like to thank Prof. Dr. Torben Weis, who agreed to act as second reviewer of this thesis. He also gave me the opportunity to present my work in a colloquium in Duisburg with a lot of fruitful discussions. I would like to thank Prof. Dr. Quang Anh Tran from Hanoi University, who also furthered my research with his valuable advice.

It has been a great experience to work with the Distributed System Group at the University of Kassel. My colleagues, Michael Zapf, Thomas Weise, Roland Reichle and Mohammad Ulah Khan, with whom I have shared my ideas and from whom I have received constructive feedback. Special thanks go to Tam Tran and Harun Baraki for their help during my stay in Kassel in May, 2015. Furthermore, I want to thank RISE students Muhaimen Shamsi and Peerapat Luxsuwong who were proofreading this thesis.

My other colleagues, Hendrik Skubch, Michael Wagner, Dominik Kirchner, Diana Reichle, Christop Evers, Daniel Saur, Stefan Niemczyk, Andreas Witsch, Tareq Razaul Haque, Stephan Opfer, Thomas Kleppe, Iris Roßbach and Heidemarie Bleckwenn were always present when I needed help and were always open to discussions. I would also like to thank Prof. Stefan Gruner and Peter Mülders from University of Pretoria, with whom I have collaborated and co-authored a publication.

Moving from individuals to organizations, I would like to thank the Ministry of Education and Training of Vietnam, the Vietnam International Education Development and the German Academic Exchange Service for awarding me the scholarship for doing my research.

Finally, but most importantly, I must thank my wife and my parents for their support, encouragement, and inexhaustible patience.
Part I
Foundation
Chapter 1  Introduction

1  Introduction

1.1  Motivation

Advances in hardware components miniaturization and wireless communications have led to the emergence of a new class of ad-hoc networks: wireless sensor networks (WSNs). A wireless sensor network connects the physical and computational world by monitoring a wide variety of environmental phenomena through devices called sensor nodes, or motes. Individual motes are often battery-powered, small in size, equipped with a variety of sensors, and have limited processing capabilities and a wireless transceiver [6, 38].

In 2003 when research and practices related to WSNs were still in the beginning stage, the highly respected journal The MIT Technology Review anticipated this technology to be one of the top ten emerging technologies that would change the world [1]. Only a few years later, the growth experienced by WSNs began to be comparable to that of the Internet revolution [173]. Many research institutions around the world have focused their efforts on topics related to WSN technology. Today WSNs have been developed for a wide range of applications for natural environments, industries, health, civil engineering or security [19, 20, 29, 34, 44, 65, 148, 155, 183, 212]. The potential WSN applications continue to increase year by year.

Despite many exciting future prospects of WSNs, experts assure that wide-spread adoption of this technology is still a future possibility. WSNs differ from conventional distributed systems in several ways. The resource limitation of sensor nodes, the ad-hoc communication and topology of the network, coupled with an unpredictable deployment environment are difficult non-functional constraints that must be carefully taken into account when designing software systems for WSNs. Sensor nodes are often equipped with a limited and nonrenewable energy source and a processing unit with a small memory capacity. Additionally, the sensor nodes and network are less reliable than those of common network systems: depending upon the configuration of network and environmental circumstances, wireless links may become degraded or unviable. In WSNs, one application may have several non-functional constraints such as power consumption and memory usage; furthermore, different applications may have different non-functional requirements and constraints. As sensor applications become more complex and diverse, accounting for these constraints becomes a more important and challenging task.

A major challenge of developing WSN applications is achieving effective programming abstractions for developing sensor software at different levels, ranging from drivers and
operating systems, to network protocols, middleware services, and applications. A WSN’s operating system is a concrete example of this problem. In a modern computing infrastructure, the main functions of an operating system are the efficient management of physical resources and the supply of high-level abstractions to simplify the programming process. Due to the limited hardware capabilities of sensor nodes, a WSN’s operating system does not always efficiently integrate the set of platforms as a whole. As a consequence, a high-level abstraction is not always feasible. As presented in [139] the abstraction level of applications with respect to the underlying levels is very low because programmers must explicitly reference the hardware and software components required by the applications. Additionally, there is no clear division in architectural layers with different abstraction levels, and there are still no well-established roles and responsibilities for each layer. Therefore, application developers must have an exhaustive comprehension of the hardware, the operating systems used, and the application logic.

Dealing with non-functional constraints, such as resource restriction and Quality of Service (QoS) are major concerns that challenge developers [35, 42, 69]. These constraints have to be considered at each stage of the system development, ranging from the operating system design to the protocol design to the applications themselves. For instance, the exact position of the sensor nodes usually cannot be determined before their deployment and it is thus not possible to ensure that each node has direct link to a base station. Therefore, the network has to be self-organizing [67, 68, 70]. The situation becomes even more complicated if such a self-organizing network becomes mobile where nodes will be able to change their geographic position, or join and leave the network arbitrarily. All these aspects are tangential to the purpose of the application itself: the collection, aggregation, evaluation, and propagation of sensor data. Although the non-functional constraints are not of primary interest to the software developer, an effort must still be made to optimize them. This process is time consuming, and thus increases the development costs significantly.

The challenges above convert applications development into a hardware-coupled task. Developers must be aware of both functional and non-functional requirements. Usually, they have to perform implementations at different layers of the software architecture: from the hardware to the high-level application itself, such as hardware driver programming and network protocols. For these reasons, the software developed for sensor nodes is typically ad-hoc and, subsequently, difficult to reuse and integrate into other systems. The consequences derived from this problem affect the development time, increase the resources involved, and delay the evolution of WSN technology.
Designing, implementing, and maintaining software for sensor nodes is a complicated and challenging task [22, 78, 97, 116, 139, 165, 185, 187]; therefore, more research contributions need to be made in this area. The motivation of this thesis is to further contribute to research being done in this area by presenting an approach to WSN application development in order to improve the reusability, flexibility, and maintainability of the software.

1.2 Problem statement

1.2.1 Problem analysis

Different aspects of the challenge of developing a WSN application have been explored in depth in several works by the research community [5, 22, 78, 139, 165, 174, 176, 187]. In this thesis, three problem sources are identified: the hardware complexity, the unique characteristics of WSNs, and the lack of specific standards, in regards to the programming interfaces and the open software architectures supporting the development process. In the following sections of this chapter these three problem sources are analyzed in depth.

1.2.1.1 Hardware complexity

The hardware complexity can be broken down into the following factors:

- The rapid growth of both prototype and commercial hardware platforms.
- The heterogeneity of the devices composing a mote, such as microcontrollers, sensors, or transceivers.
- The restricted capabilities of the motes such as memory, CPU cycles, and transmission rate.

New hardware platforms for sensor nodes are continuously developed by hardware manufacturers, resulting in enhancements such as improved power-efficiency in microcontrollers, modern sensors, or alternative batteries to extend the sensor node’s lifetime. According to a hardware survey maintained by the researchers at the Imperial College London [119], more than fifty WSN mote platforms have so far been produced. The high number of available platforms is reasonable because, as pointed out in [175], such diversity has been partially supported by the wide range of applications requirements. Figure 1.1 shows the evolution of the number of mote platforms in the last decade, and Table 1.1 demonstrates the diversity of commonly used platforms.
### Table 1.1 An example of mote platforms diversity

<table>
<thead>
<tr>
<th>Platform</th>
<th>Year</th>
<th>Microcontroller</th>
<th>Memory</th>
<th>Serial comm.</th>
<th>Wireless com.</th>
<th>Programming &amp; OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MicaZ</td>
<td>2004</td>
<td>ATMega 128 8 bit</td>
<td>128K Flash 4K RAM 512K ROM</td>
<td>UART</td>
<td>TI CC2420 802.15.4/ ZigBee compliant radio</td>
<td>nesC &amp; Tiny OS</td>
</tr>
<tr>
<td>Imote 2</td>
<td>2007</td>
<td>Intel PXA271 16 bit</td>
<td>32M Flash 32M RAM</td>
<td>USB/UART T/GPIO/ I2C</td>
<td>TI CC2420 802.15.4/ ZigBee compliant radio</td>
<td>C, nesC Microsoft.NET Micro, Linux, TinyOS Support</td>
</tr>
<tr>
<td>SHIMMER</td>
<td>2008</td>
<td>MSP 430 16 bit</td>
<td>48K Flash 10K RAM microSD support</td>
<td>Integrated sensors</td>
<td>TI CC2420 802.15.4 Bluetooth support</td>
<td>nesC &amp; Tiny OS</td>
</tr>
<tr>
<td>iSense 2</td>
<td>2009</td>
<td>RISC 32 bit</td>
<td>512K Flash 128K RAM</td>
<td>UART/I2C</td>
<td>Jennic JN5148 802.15.4/ ZigBee compliant</td>
<td>C++ modular operating and networking firmware</td>
</tr>
<tr>
<td>Arduino BT</td>
<td>2010</td>
<td>ATMega 328 8 bit</td>
<td>32K Flash 2K SRAM 1K ROM</td>
<td>UART/PWM I2C</td>
<td>Bluetooth</td>
<td>Arduino language is based on C/C++</td>
</tr>
<tr>
<td>Zolertia Z1</td>
<td>2011</td>
<td>MSP 430 16 bit</td>
<td>96K Flash 8K RAM</td>
<td>UART/I2C</td>
<td>Chipcon CC2420 802.15.4 compliant</td>
<td>C, nesC Contiki &amp; TinyOS support</td>
</tr>
<tr>
<td>Wasp Mote Pro</td>
<td>2012</td>
<td>ATMega 1281 8 bit</td>
<td>128K Flash 8K RAM 4K ROM microSD support</td>
<td>UART/I2C/USB/SPI</td>
<td>3G/GPRS/802.15.4/ ZigBee/ Bluetooth/RFID/ NFC</td>
<td>C with WaspMote IDE Wasp firmware</td>
</tr>
</tbody>
</table>

The three factors make the application development cumbersome. Firstly, the resource constraints affect the architecture of an application by determining the maximum...
amount of available space (data and code) that the application can allocate. Secondly, every new platform must be supported by a WSN operating system to operate, but currently there is no single operating system that supports all existing hardware platforms. Last but not least, the hardware complexity also limits the application’s portability and reusability. Portability refers to the ability of an application to be used in different environments. In contrast, reusability refers to using components of one application unit to facilitate the development of a different application with a different functionality [137]. Due to the diversity of hardware in WSNs, the effort of porting (and reusing) an application to new platforms is dependent on the novelty of the platform itself, and usually increases the workload of hardware experts and software developers.

The hardware complexity influences the operating system’s functionality, and the operating system itself imposes restrictions on the applications. In WSNs, the operating system determines the way applications are programmed. Applications must be written in the same programming language as the underlying operating system. For example, TinyOS applications must be written in nesC [113], and Contiki and Mantis applications must be written in the C programming language. Furthermore, there is a strong coupling between the programming style and the execution model supported by the operating system. In fact, the execution model is not kept hidden and determines the programming model. Consequently, the same application looks very different, depending on whether it has been programmed in Contiki, TinyOS, or other operating systems. Table 1.2 demonstrates a variant of the Blink application for different operating systems. The Blink application causes the red LED on the mote to turn on and off at 1Hz.

<table>
<thead>
<tr>
<th>Blink for Tiny OS 1.x</th>
<th>Blink for Tiny OS 2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>module BlinkM {</td>
<td>#include &quot;Timer.h&quot;</td>
</tr>
<tr>
<td>provides {</td>
<td>module BlinkC @safe() {</td>
</tr>
<tr>
<td>interface StdControl;</td>
<td>uses interface Timer&lt;TMilli&gt; as Timer0;</td>
</tr>
<tr>
<td>}</td>
<td>uses interface Leds;</td>
</tr>
<tr>
<td>uses {</td>
<td>uses interface Boot;</td>
</tr>
<tr>
<td>interface Timer;</td>
<td>implementation {</td>
</tr>
<tr>
<td>interface Leds;</td>
<td>event void Boot.booted() {</td>
</tr>
<tr>
<td>}</td>
<td>call Timer0.startPeriodic( 1000 );</td>
</tr>
</tbody>
</table>
| }                   |     }
| implementation {    | event void Timer0.fired() { |
|     command result_t StdControl.init() { |     call Leds.led0Toggle(); |
|         call Leds.init();  |     }
|         return SUCCESS;   |     }
|     command result_t StdControl.start() { | }
|         return call Timer.start(TIMER_REPEAT, 1000); |
|     command result_t StdControl.stop() { | }
|         return call Timer.stop(); | }
Blink for Contiki OS

```c
PROCESS(blink_process, "Blink");
AUTOSTART_PROCESSES(&blink_process);
PROCESS_THREAD(blink_process, ev, data) {
    static struct etimer et;
    while (true) {
        leds_toggle(LED_RED);
        etimer_set(&et, CLOCK_SECOND);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    }
    PROCESS_END();
}
```

Blink for Mantis OS

```c
void blink_thread(void) {
    while(1) {
        mos_led_toggle(0);
        mos_thread_sleep(1000);
    }
}
```

### Table 1.2 Blink application for different operating systems

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Code</th>
</tr>
</thead>
</table>
| Contiki OS       | ```c
PROCESS(blink_process, "Blink");
AUTOSTART_PROCESSES(&blink_process);
PROCESS_THREAD(blink_process, ev, data) {
    static struct etimer et;
    while (true) {
        leds_toggle(LED_RED);
        etimer_set(&et, CLOCK_SECOND);
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    }
    PROCESS_END();
``` |
| Mantis OS        | ```c
void blink_thread(void) {
    while(1) {
        mos_led_toggle(0);
        mos_thread_sleep(1000);
    }
``` |

In conclusion, the software for sensor nodes has been built following a bottom-up strategy because of the hardware complexity. Consequently, applications are frequently tailored to specific hardware and an operating system. In many cases, due to the strong coupling between the application and the underlying operating system, portability and reusability are sacrificed.

#### 1.2.1.2 Unique characteristics of WSNs

The basic function of a WSN is to perform networked sensing using a large amount of unsophisticated and inexpensive sensors. The potential benefit of WSNs compared to other approaches can be summarized as having greater coverage, accuracy and reliability at a possibly lower cost [5, 38, 165, 176, 222]. This concept of WSNs implies a set of unique characteristics which heavily influence the application development process. These characteristics are outlined as the following:

- The high degree of network dynamics caused by node failures, an unreliable wireless communication link and environmental obstructions.
- The application-specific architecture where the details of the application determine the optimization of lower-layer protocol stacks.
- The cooperation among motes to accomplish their tasks and to achieve a common goal.

Most WSNs consist of a large number of motes, ranging from hundreds to thousands, which are deployed in a high density. The reason is that with a large number of nodes and a high node density a WSN will be more robust to routing and node failure because
of redundancy in the network. In order to save energy motes are often scheduled to sleep at periodic intervals. In this case, motes may join and leave the network from time to time, and as a consequence, the network topology is almost impossible to maintain. Additionally, the short range unreliable wireless communication link between motes is also a contributing factor in the network dynamics.

In many WSN applications, motes are sometimes randomly deployed in remote or inaccessible areas, after which the sensor network must operate without human intervention. This means that network configuration, adaptation, maintenance, and repair must be performed in an autonomous fashion. For example, sensors to survey a battlefield could be dispersed from airplanes over areas of interest, but many motes may be damaged from the fall, and might not even startup. However, the surviving motes must autonomously perform the configuration procedure and carry out their tasks.

Due to the high degree of network dynamics, WSN protocols and algorithms must possess localization, scalability and self-organizing capabilities that enable them to autonomously adapt to changes during their operation. Localization in this context means each node makes decisions solely based on the awareness about its local neighbors in order to minimize the number of messages between nodes, to conserve bandwidth and energy.

Aside from network dynamics, application specific architectures are another unique characteristic of WSNs. It is clear from the survey of sensor network applications [38, 120, 175, 221] that WSNs are designed for specific applications. Such applications include, but are not limited to, environment monitoring, object surveillance and tracking, and building facility management. Each application differs in features and QoS requirements, and these application requirements determine the architecture and the communication protocols of the sensor network. To illustrate this, let us consider the following two scenarios. First consider an application to monitor the environmental conditions of the petrels nest [122]. This is a data gathering application. The structure of the generated data should be uniform, and the latency requirements on the application’s data are expected to be loose. In this case, the challenge is to find and maintain energy-efficient routes to convey all the collected data to the base station. On the other hand, a sensor network deployed to detect forest fires [80] would occasionally generate data with severe latency constraints. Therefore, maintaining routes between sensor nodes would not be energy-efficient. In this case, it would be important to route the data promptly, but with a reasonable cost of route maintenance.
1.2 Problem statement

The two scenarios discussed above do not exist in traditional communication networks because these networks aim to support a diverse set of users, each with their individual objectives [126]. For example, the internet is modular and interoperable with a layered protocol architecture that can support new application on top of the network. Consequently, the tradeoff between application requirements such as QoS constraints and the resource limitations of the sensor nodes has been unfound in traditional communications and networking. The traditional layered architecture of the communication protocol stack has also been identified as insufficient in addressing the new challenges, where non-traditional design principles like component-based design, cross-layer integration or application-specific design are needed [43, 102, 103, 129, 172, 175]. Typically, designing the application-specific protocols for WSNs could be a solution, but the application-specific approach may lead to the development of different protocols for each application. Hence, the research and development being done on WSNs stand to benefit from a flexible software framework that can be customized and adapted to suit different applications.

The cooperation among motes in a network is the third unique characteristic of a WSN which is explored in this thesis. WSNs provide perfect environments for sensor collaboration to fulfill a global purpose. For instance, due to the limit on its communication range, each node must cooperate to transmit data throughout the network using a multi-hop protocol. Additionally, nodes may collaborate to perform in-network processing in order to optimize communication bandwidth and error rates. In-network processing (i.e., data aggregation) represents intermediate network computing amongst localized nodes to improve energy efficiency and data delivery performance by reducing network traffic load. Collaborative in-network processing can be a very challenging requirement when programming WSNs applications. It promotes the use of distributed algorithms where nodes interact with their neighbors to solve a given task efficiently [60, 72, 144, 166, 207, 209]. However, in practice it is difficult to devise such algorithms. In order to support the design and implementation of the distributed algorithms, WSN developers need a programming model and a framework that provides an easy way of describing the application logic and hiding the complexity that originates from the distributed and unreliable nature of the WSN.

1.2.1.3 Lack of software architectures, supporting tool and specific standards

In several works related to WSN architecture [15, 16, 37, 101, 135, 218], the authors have concluded that the lack of an overall architecture is one of the primary factors currently limiting progress in WSNs. Despite the fact that WSN motes have limited resources, they are more or less networked computers. Thus, an appropriated software architecture which provides support for distribution, configuration, scalability, and
portability is necessary for the WSN’s development. The functional requirements of such architectures are illustrated in Figure 1.2(a), from the point of view of the developer.

The range of the functional requirements may be broadly classified into three groups. The first group is the system management. Because each sensor node is an individual system, to support different application software on such a sensor system, the development of new platforms, operating systems, and storage schemes are needed. The second group is communication protocols, which enable communication between the application and sensors, as well as the communication between the sensor nodes. The last group is application supports which are developed to enhance the application and to improve system performance and network efficiency.

Functional requirements are partially fulfilled by several distinct architectures and operating systems in different research projects. For example, TinyOS and component-based architecture [113] are a very popular solution. Conceptually, a component-based architecture consists of a scheduler and a set of components connected to each other through well-defined interfaces. Although TinyOS does not provide a clear separation between the operating system layer and the application layer, the logical boundary between them are illustrated in Figure 1.2(b). The component-based architecture in TinyOS is compact and resource-efficient; however, this architecture still presents several disadvantages such as the difficulty in maintaining or updating the application, since it is statically linked to the whole kernel. Moreover, invoking hardware components directly from application modules is a widely used programming technique.
1.2 Problem statement

in TinyOS, and other aspects like security, routing, or network reconfiguration must be implemented in the application layer.

Other trend in the research of WSN software architecture is the construction of a middleware to support the development process [78, 135, 176, 214, 218]. Figure 1.3 shows a generic architecture in which the middleware sits between the operating system layer and the application layer in order to satisfy the application specific requirements. In general, middleware provides an integrated solution to meet the application requirements, hide details of lower levels, and facilitate the development, deployment and management of applications. However, we believe that a monolithic integrated solution is not a good idea, because it leads to reduced flexibility, and therefore integration could be delayed. A heavily integrated solution is also difficult to maintain and update as well. An example is TinyDB [121], despite its popularity for TinyOS 1.x, there is still no TinyDB for TinyOS 2.x.

Beside an appropriate software architecture, the availability of methodologies and tools that support users to abstract low-level details, develop WSN software, deploy and operate the software on the network will greatly contribute to the commercial success of WSN applications. For example, the development of software would be much simpler if we had a programming environment for WSNs in which the syntax was corrected, and it could compile a program and test it in a simulator that could simulate a single mote or a set of interconnected ones. Such the development environment for PCs instead of
WSN can be Eclipse (www.eclipse.org), Visual Studio (www.microsoft.com/visualstudio), or other development platforms that provide a range of tools offering various benefits to developers. Although there are some proposals regarding software design for WSNs, so far, there have been no adequate tools or frameworks to aid application development, especially in terms of the investigation and definition of methodologies for generating reliable software.

Finally, while traditional computer networks are based on established standards, many protocols and mechanisms in wireless sensor networks are proprietary solutions, and the standards-based solutions emerge very slowly. Standards are important for interoperability and facilitate the design and deployment of WSN applications; therefore, standardization will continue to be a key challenge in WSN.

1.2.2 Research objective

This thesis addresses the above mentioned challenges and establishes two main objectives as the following:

- Design, implement and evaluate a framework for developing WSN applications which can be easily ported and reused among different mote platforms.
- Identify and determine application-specific requirements for WSNs application as the non-functional constraints and factors that should be considered when developing the applications. Define an appropriate methodology and implement supporting tools for investigating and optimizing these constraints as a part of the framework.

More specifically, the main objectives above are split up in the following specific objectives:

1. Design a multi-layered software architecture clearly distinguishing the different abstraction levels:
   - At the hardware level, a reasonable set of physical devices have to be studied in order to elaborate a generic and flexible hardware model, including the analysis of the resources and their properties, functionalities, and services.
   - At the operating system level, several operating systems for sensor nodes will be considered and analyzed including TinyOS (versions 2.x) and Contiki. As in the previous case, a generic model describing this level will be stated.
1.3 Research method

- An abstraction layer for heterogeneous WSNs operating systems will be proposed and implemented. This intermediate layer is intended to provide a standardized access to the underlying architecture.
- At the application layer, a Domain-Specific Language (DSL) has been created to support the development of portable applications on top of the previous layer. The DSL also enable the modeling of the application-specific requirements as the non-functional resource constraints.

2. Design and implement an experimental framework that adopts the architecture above and supports to:
   - Interpret and transform the DSL into a lower level language
   - Compile and generate the program code
   - Simulate the deployment of the developed application

3. Define a reusable methodology for addressing the problem of optimizing non-functional constraints with the implementation of the supporting tool as a part of the experimental framework.

4. Demonstrate several test cases and evaluating the performance of the approach based on experimental statistical data.

1.3 Research method

Based on the problem analysis, as well as the identified research objectives and goals, we adopt the design method [170] for the work presented in this thesis. The design method consists of four steps that are followed in the development of a system to tackle a given problem:

i) identifying requirements
ii) stating specifications
iii) designing and implementing the system
iv) testing the developed system

At the first step, we studied the hardware and software architecture of a generic mote, and analyzed the physical devices to extract their settings, functionality, and interface. We went into detail on several operating systems specifically designed for motes, in order to explore and compose a flexible software architecture which is sufficient enough to deal with the heterogeneity and applications portability.

At the second step, we investigated the state of the art in WSNs programming model including programming approaches, platforms, languages, as well as their pros and cons. The taxonomy of WSN applications was also studied and presented in this thesis in order
to identify, capture and represent the application’s specific requirements as the non-functional constraints that need to optimize. Then, we proposed a set of specifications and criteria, such as appropriate software architecture, development model and other supporting techniques. Several different optimization techniques were also investigated and discussed, focusing much on multi-objective optimization solutions in order to optimize these constraints.

Afterwards, a real prototype framework was built to demonstrate the developed solutions in the project. The results were compared to state-of-the-art solutions, and the developed prototype integrated the accumulated knowledge from the previous steps.

Finally, we tested and evaluated the prototypes in terms of technical feasibility, strengths, ease of use and resource efficiency using the simulation. During the working period, research papers have been peer-reviewed by experts in the field. Presentations of the research results at international conferences have also provided relevant feedback and opportunities for improving our proposals and exchanging ideas with other researchers.

1.4 Solution approach

This session provides an overview of the proposed solution to the aforementioned problems. Firstly we present an architecture aimed at describing WSN applications with independence to the underlying operating system and hardware. Then we explain, step by step, how the proposed architecture can be used for WSN application development. The optimization process is also described.

The proposed software architecture for WSN mote is illustrated in Figure 1.4(a). The architecture is designed in a multi-layered approach, integrating the traditional architecture of WSN mote and two upper layers: the Application Layer (APL) and the Operating System Abstraction Layer (OAL). The goal of the upper layers is to offer a higher level of abstraction and portability when hiding the lower platform. The APL is intended to provide the required abstractions and establish the writing rules to program the applications, for this purpose, a Domain Specific Language (DSL) is designed to support isolating the application from the operating system and constructing the sensor node program. The goal of the OAL is to translate generic applications written using the DSL into the equivalent operating system-specific implementation. Figure 1.4(b) demonstrates different examples of the proposed architecture, in which the different components at each layer are presented. A concrete mote’s software architecture could be formed by combining these components, one unit in each layer.
The requirements of separation of concerns, hiding low-level details and raising the level of abstraction lead to a solution offered by Model-Driven Development (MDD) approach (refer to Chapter 3 below). Among the MDD approaches, the Model-Driven Architecture (MDA) developed by the Object Management Group (OMG) is the most promising and widely known one. Models are the primary artifacts in MDA; there are three types of models representing three levels of abstractions: the Computation Independent Model (CIM), the Platform Independent Model (PIM) and the Platform Specific Model (PSM). CIM is a view of a system that abstracts away the computation requirements. PIM is a platform independent view of a system, and PSM is the most concrete model that combines the PIM with specific details of a particular platform. In an MDA approach, the specific system architecture is created through a process of transformations of the abstract models (CIM, PIM) into the concrete system designs (PSM).

Following this approach, we employ the MDA standard to describe and realize the proposed architecture. To start, we assume that application logic and application requirements have been previously defined, and the application’s CIM was already created. At the APL, a PIM is used to capture the application’s logic and requirements. A DSL is designed at this level of abstraction to support the mapping process. At the OSL, a set of OS-specific platform models corresponding to the different WSN operating systems such as TinyOS or Contiki is presented. Then, given an OS-specific platform model, a transformation process is defined to translate the PIM to a PSM. At this level of abstraction, the PSM is OS-specific, and may use a general purpose language like C or
nesC [113] which is supported by the target operating system. The OAL carries out the transformation of the PIM to the PSM; therefore, this layer can be viewed as an application generator, which produces the tailored code to deploy in the target platform. The lower layers (HAL and HL) consist of a compiler, a simulator, the OS-specific libraries and the hardware-specific PSM that provide the interfaces to the concrete mote hardware platforms. A hardware-specific PSM describes in detail the hardware components which compose a concrete mote platform like MicaZ or Zolertia Z1. This PSM is used by the compiler to generate the binary code from the program code received from the upper layer. A simulator is developed and integrated in order to provide the simulated motes and networks to test and evaluate the generated solution.

As mentioned above, one critical WSN design challenge involves meeting the non-functional constraints such as lifetime, reliability, throughput, delay, etc. In addition, WSN applications tend to have competing requirements, which exacerbates the design challenges. In this thesis, we provide approaches to optimize these non-functional properties in the context of WSN. In order to do so, on one hand, we define a set of application’s adjustable parameters whose values can be specialized to meet application requirements. These parameters assist designers in customizing the application at different design levels, such as hardware (processor voltage and frequency), software (sensing frequency, duty cycle), or communication protocol (channel access schedule, message size, buffer size, and receiver power-off cycle). On the other hand, we figure out how the non-functional properties can be qualitatively specified and quantitatively measured in the context of WSN, as well as how the non-functional constraints can be expressed by defining an objective function to maximize or minimize a set of non-functional properties. In this way, the optimization problem means to find the best set of application adjustable parameters which satisfy the objective function, from the search space of all feasible parameter values.

Because of the complex interrelationships between parameters, the large number of their possible values and the measurable responses, the optimization problem is not simple or idealized; actually it is a real-world problem with multiple inputs, multiple outputs, and multiple objectives. In order to deal with this problem, two different approaches are explored in this thesis. The first approach is the Systematic Statistical Design of Experiments (SSDOE) [24, 197] which is a well-known experiment design theory. SSDOE is applied to gain a better understanding of the potential relationship between input variables and output responses. This approach requires the identification of factors (inputs to an experiment with different values or levels) and responses (outputs of the experiment, observations or measures). A series of experiments is run with all possible combinations of all acceptable values of the factors and the responses
are recorded. Then statistical methods are used to identify the significant factors or combinations of factors that impact the response of interest, and to produce a statistical model to predict the response values. The SSDOE could be considered as a broad but shallow search method which explores the search space in a systematic and even manner. The main disadvantage of SSDOE method is that the experiment cost will grow rapidly when there are many factors, and each factor has many acceptable values.

Alternative approaches, which are better suited to large numbers of input factors, are considered in the thesis. For example, an evolutionary algorithm (EA) could be a feasible candidate since all factors can be mutated at each generation in EA. We apply a Multi-Objective Evolutionary Algorithm (MOEA) [45, 62, 223] to solve the optimization problem. The MOEA could be considered as a narrow but deep heuristic search method which explores the search space in an uneven manner. An optimization component based on SPEA2 [224] is integrated in our framework to support developers to optimize their applications in terms of non-functional constraints.

Figure 1.5 presents eight steps in the application development process using our MDD framework. These steps can be summarized as follows:
• Step 1 – Modeling: Starting with an application’s requirement specification, a
PIM, which consists of the formal definition of the applications, is created using a
DSL. The proposed DSL offers a set of declarative sentences to express the
behavior of sensor nodes such as sampling, aggregation and forwarding which is
necessary for developing WSN applications. A PIM also contains other
information that will be used in the later steps, such as the set of application’s
adjustable parameters, the non-functional properties, the objective function as
well as the simulation configuration.

• Step 2 – Transforming: The transformation process consists of the
implementation of mapping rules and code template files to transform PIM into
the equivalent operating system-specific code (OS specific PSM).

• Step 3 – Compiling: The binary code (Hardware specific PSM) for the selected
hardware platform is generated.

• Step 4 – Testing: The binary code is deployed and tested on the simulator with
the pre-defined simulation configuration in the PIM. The simulator has two
interfaces. The first one is a friendly graphical user interface, which is used to
visualize the output of the simulation process. The second one is a text-based
interface, which is used to provide normalized measured data to the
optimization component. Depending on the testing result the developer will
determine whether the application satisfies all the functional requirements.

• Step 5, 6, 7 – Optimizing: In these steps the successfully tested application is
optimized. The set of adjustable parameters, which is defined in PIM, is used by
the optimization component to generate the search space (step 5). The
evaluation of each candidate solution (step 6) is done by: i) repeating from step 2
to step 4; ii) evaluating the normalized measured data from the simulator (step
7).

• Step 8 - Choosing: Based on the optimization results the developer will select the
optimal application by evaluating the tradeoff between different constraint
criteria and performance value.

1.5 Structure of the thesis

This thesis is structured into three parts. The foundations of the proposed solution are
discussed in the remainder of the first part. This includes relevant aspects of WSN
technology (Chapter 2), model-driven approach, and non-functional requirements
(Chapter 3), as well as the analysis of related work (Chapter 4).

The main contribution of this thesis is contained in Part II of the document. In Chapter 5,
we analyze current programming models in WSNs and introduce a rule-based
programming model that is implemented in our framework. The proposed software
architecture for WSN motes with different abstraction levels is described in Chapter 6. Chapter 7 presents the proposed DSL and application modeling. The taxonomy of WSN applications is also investigated and presented in this chapter in order to identify, capture and represent the application’s specific requirements as the non-functional constraints. Chapter 8 describes transformation and code generation tools. The simulator and measuring non-functional properties are presented in Chapter 9. In Chapter 10, we discuss two optimization approaches SSDOE and MOEA.

In Part III of the thesis, we evaluate and critically assess the proposed concepts. Two case studies are provided to support the evaluation including the framework evaluation and the development and optimization of a real-world application using the proposed framework. The goal of framework evaluation is to assess the ease at which novice and intermediate users can develop correct and power efficient WSN applications, the portability level achieved by developing applications at a high-level of abstraction, and finally estimate the overhead due to the framework usage in term of the footprint and executable code size of the application. The details of framework evaluation are presented in Chapter 11. Chapter 12 discusses the design, implementation and optimization of a real-world application named TempSense, where a sensor network is used to monitor the temperature in an area. In our experiment, we designed and implemented TempSense on a target platform (Z1 and ContikiOS) using the proposed framework. Then we optimized the application in terms of power efficiency and reliability with two proposed approaches SSDOE and MOEA. Finally, Chapter 13 concludes this dissertation and presents an outlook to future work.
2 Wireless sensor network

2.1 WSN architecture and hardware technology

2.1.1 Network architecture

Sensor nodes (or motes) are small computational devices which have gained a lot of interest in research and industry during the last decade. The potential of these devices is based on the integration of different sensors, from simple sensors (i.e., temperature, light, humidity) to complex sensors (i.e., GPS, imagers, micro radars), which make it possible to measure a wide range of environments and to monitor physical phenomenon, providing users with accurate and up-to-date knowledge. Other than basic sensors, sensor nodes are equipped with a central processing unit (CPU), some working memory, storage memory, and communication facilities — so basically forming small networked computers. Sensor nodes form networks mostly based on wireless communication called wireless sensor network (WSN).

A WSN may integrate several hardware entities. Figure 2.1 shows the architecture of a typical WSN, where the following elements can be clearly distinguished: mote (M), sensor device (S), actor (A), and gateway (G).

![Figure 2.1 WSN architecture](image)

A sensor node is composed of a mote and several sensor devices. A mote is the entity composed of a processor and radio devices. Sensor devices attached to the mote...
through an expansion connector, sometimes the sensors are integrated into the mote itself that is why sensor node is usually referred to as “mote”. Some examples of sensors are temperature, humidity, accelerometer, magnetometer, or sound. A WSN may include a gateway which provides the interface between the wireless network and the external world. For instance, a typical gateway incorporates a RS-232 or USB port, which is used to forward data to the PC, or an Ethernet port to forward data to the Internet. Optionally, a WSN may consist of actor nodes which take decisions and then perform appropriate actions upon the environment [5, 111]. The actor nodes are heterogeneous devices including robots, unmanned aerial vehicles, and networked actuators such as water sprinklers, pan/tilt cameras, robotic arms, etc. Finally, a WSN consists of a sink node, which is sometimes referred to as a base station. In general, the sink node, where the collected data is passed on to, could belong to the WSN or be part of an external network (i.e., internet). In the second case, it is called a base station, and connected to WSN through a gateway node.

2.1.2 Hardware technology

Usually sensor nodes integrate a low-power microcontroller including RAM (for data) and ROM (for code) memory chips of small capacity, an analogical-digital converter (ADC), and a radio transceiver which allows sending and receiving data to or from other similar devices. Different kinds of sensors can be attached directly to the device or connected to the mote through an expansion connector. The energy source typically consists of conventional batteries, solar sources, temperature differences, or vibration.

Figure 2.2 presents the generic hardware architecture of a WSN mote. As shown, it is composed of a set of hardware components which are described as follows:

**Micro-controllers:** they usually are low-capacity processors such as Atmel Atmega 128L [14] or MSP430 [94] which operate at low frequencies, and range from 4-bit to 32-bit architectures. They also contain different RAM and ROM memories, an Analogical-Digital Converter and different clocks in order to allow local timing. Two examples of popular microcontrollers are:

- **Atmel Atmega 128L [14]:** from ATMEL, a low-power CMOS 8-bit microcontroller based on the RISC architecture. This microcontroller has 32 general purpose working registers. The memory system is composed of 128 KB of In-System Programmable Flash, 4 KB EEPROM, and 4 KB SRAM. The system clock is composed of several clocks allowing four timers: two 8-bit and two 16-bit width timers. The micro also incorporates a 10-bit ADC. The microcontroller supports six software selectable energy saving modes.
- MSP430 [94]: from Texas Instruments, an ultralow-power CPU, which has a 16-bit RISC architecture. In the lowest of the five power modes the power consumption is about 0.4 μA. From there, the MSP430 can switch in less than 1 μsec into its active mode. For the operation of the real-time clock it needs about 2 μA. The MSP430 knows 27 basic instructions, 24 emulated instructions, and 16 prioritized interrupt sources. Clock speeds are available up to 4 MHz. Memory sizes amount up to 60 KB ROM and 2KB RAM. The clock system is supported by the basic clock module that includes support for a 32768 Hz watch crystal oscillator, an internal digitally-controlled oscillator (DCO) and a high frequency crystal oscillator. The modules are capable of generating 16-bit timers. The ADC12 is the module responsible for performing 12-bit analog to digital conversions at the very high rate of 200 thousand samples per second.

**Transceiver:** This device provides wireless communication to the sensor node. Some transceiver devices for motes are the following:

- TR1000 [136]: this transceiver is manufactured by RF Monolithics, and it was the first radio device integrated into a sensor node. The main feature is its low consumption both for transmitting and receiving, especially in the sleep mode. TR1000 uses two frequency bands: 868 and 916 MHz.
- CC1000 [162]: the transceiver from Chipcon operates in a frequency selectable from 300-1000 MHz, but is mainly intended for 315, 433, 868 and 915 MHz frequency bands. Low power consumption is one of the most attractive characteristics of this device. This transceiver is used in Mica2 and Mica2dot motes.
2.2 Communication protocols and standardizations

- **CC2400 [163]:** This device is one of the first radio Zigbee-compliant integrated into Crossbow devices. This radio device achieves a data rate of 250 Kbps (which means 6.5 times more than its predecessor), but as a disadvantage, its consumption is notably higher. This device is used in MicaZ and Telos family motes.

- **nRF2401 [178]:** The transceiver from Nordic Semiconductors allows transmitting and receiving at the 2.4-2.5 GHz ISM band, and therefore is ZigBee compliant. One of its advantages is that it can achieve the highest data rate to date (1 Mbps), in addition to its low power consumption. This device is used in the Arduino and Sensor Cube [49] platforms.

*External memory:* They are external flash chip memories with higher capacity than the internal memories, capable of storing temporal data provided by different sources (sensors, network or logs).

*Expansion connector:* They provide interfaces for both sensor boards and gateways. The connector includes interfaces for power and ground, power control of peripheral sensors, ADC inputs for reading sensor outputs, UART and I2C interfaces, general-purpose digital I/O, and others. The connector can be USB, 51-pin or 19-pin.

According to a hardware survey maintained by the researchers at the Imperial College London [119], more than fifty WSN mote platforms have been produced until now. New hardware platforms for sensor nodes are continuously manufactured by the hardware manufacturers including the enhancements such as more power-efficiency microcontrollers, modern sensors, or alternative batteries to extend the sensor node lifetime. Table A.1 (Appendix A) describes in detail the sensor nodes evolution in a timeline from 2002 to 2012.

2.2 Communication protocols and standardizations

Communication among sensor nodes is a critical issue for several reasons. On one hand, it is a mandatory functionality since it allows each sensor node to communicate with others and to the sink. On the other hand, the several hardware restrictions make impracticable some conventions employed both in internet and mobile networks. We review the various energy-efficient protocols proposed for the data-link layer, network layer, transport layer, and communication standards in the following subsections.

2.2.1 Physical layer

The physical layer provides an interface for transmitting bit streams over the physical communication medium. It is responsible for interacting with the higher layer - MAC
layer, performing transmission and reception, and modulation. In a WSN, the energy-efficient design should start at the physical layer. At this layer, energy is used in operating radio circuitry and bit stream transmission. In general, the energy used to transmit the data can vary depending on the transmission distance, channel loss, and the adjacent channel interference, whereas the energy spent to operate the radio circuitry is fixed. There is a tradeoff between transmission power and error. Hence, proper selection of the transmission power is needed to minimize energy loss and for the network to operate more efficiently.

In the physical layer, modulation schemes are needed to transmit data over a wireless channel. Different modulation schemes have been developed to achieve the highest probability of successful transmission under different conditions. Energy-efficient modulation schemes should minimize both transmission and circuit energy. Recent research [6, 182, 215, 221] discusses the physical layer requirements, bandwidth selection, low power radio architecture design, power-aware transmission schemes, and modulation schemes.

**Bandwidth choices:** there are three classes of bandwidth in WSN: narrow band, spread-spectrum, and ultra-wideband. Narrow band uses radio bandwidth that modulates all of the data upon a single carrier frequency. Hence, narrow band focuses on bandwidth efficiency which is the measure of the data rate over the bandwidth. In spread-spectrum, the narrow signal is spread into a wideband signal. The spreading function used to determine the bandwidth is independent of the message. Spread-spectrum has the ability to reduce power and still communicate effectively. It is more robust to interference and multi-path channel impairment. Compared to spread-spectrum, ultra-wideband employs larger bandwidth, on the order of gigahertz, compared to the typical spread-spectrum systems. Ultra-wideband spreads its signal over the large bandwidth such that the interference to other radios is negligible. Like spread-spectrum, ultra-wideband can communicate with low power.

**Radio architecture:** The energy consumption at the physical layer could be decreased by reducing the circuitry energy and transmission energy. In order to save energy in WSN, motes often have a duty cycle mechanism to power down the transceiver during idle times. To start a transmitter, a significant amount of time and energy is required. Energy at startup in some cases can be higher than the energy required for an actual transmission. For a transmitter that switches between the sleep to active state, the fast startup transmitter architecture is needed to minimize both energy and time.

- FN frequency synthesizer with ΣΔ modulator [206]: this architecture based on a fractional-N (FN) frequency synthesizer with ΣΔ modulator. The architecture
2.2 Communication protocols and standardizations

achieves fast startup time and data rate by increasing the loop bandwidth. Each noise source from the synthesizer and the modulator goes through different loop characteristics. By adjusting the loop bandwidth, power consumption can be reduced.

- WiseNet [58]: This architecture also seeks to reduce power consumption with low voltage operations. WiseNet employs a dedicated duty-cycle radio and a low power MAC protocol design (WiseMAC) to lower its power consumption. To optimize the startup time and save energy, the system wakes up the different transceiver blocks in a sequence. The lower-power baseband blocks wake up before the radio frequency (RF) circuits. Startup time varies inversely with the frequency of operation.

Modulation scheme: Energy-efficient modulation schemes are needed to reduce energy consumption. Common modulation scheme are used in WSN including binary modulation and multi-level (M-ary) modulation. In [182, 215] the authors presented a comparison between two modulation schemes. M-ary modulation transmits symbols from a set of M distinct wave forms while binary modulation uses two distinct waveforms. For M-ary modulation, \(\log_2 M\) bits are sent per sample. It is shown that M-ary modulation is more energy efficient than binary modulation when the startup time is short and the RF output power is small. There are various types of M-ary modulation including M-ary frequency shift keying (M-FSK), M-ary phase shift keying (M-PSK) and M-ary quadrature amplitude modulation (M-QAM). For a large value of M, M-FSK is more energy efficient compared to M-PSK and M-QAM. For small M, M-FSK is not as energy efficient because more RF power is required to achieve the same bit-error-rate performance as M-PSK and M-QAM.

In conclusion, Table A.2 (Appendix A) summarizes the design requirements of the physical layer in WSN, as well as the available solution and their main characteristics.

2.2.2 Data-link layer

The data-link layer is concerned with the data transfer between two nodes that share the same link. Since the underlying network is wireless, for effective data transfer, there is a need for medium access control and management (MAC). As pointed out in [47, 208], to design a good MAC protocol for wireless sensor networks, the following attributes must be considered: energy efficiency, scalable to node density, frame synchronization, fairness, bandwidth utilization, flow control, and error control for data communication. A wide range of MAC protocols have been proposed to meet these requirements. In this session, we review some of the representative approaches.
2.2.2.1 Sensor-MAC (S-MAC)

S-MAC [208] protocol has been implemented in TinyOS for Mica, Mica2 and Mica2dot motes to reduce the energy waste caused by idle listening. S-MAC is a Carrier-Sense Multiple Access (CSMA) protocol. In S-MAC, sensor node periodically goes to the fixed listen/sleep cycle. The listen/sleep schedule requires synchronization among neighboring nodes and updating schedules is accomplished by sending a SYNC packet. A time frame in S-MAC is divided into two parts: one for a listening session and the other for a sending/receiving/sleeping session. In the listening session, nodes communicate with each other and send some control packets such as SYNC, Request to Send (RTS), Clear to Send (CTS). A SYNC packet is exchanged between all neighbor nodes to synchronize. The RTS/CTS packets are exchanged between two nodes which want to send/receive data in the next session. Collision avoidance is achieved by using carrier sense (CS) technique. In the sending/receiving/sleeping session, the sending node and receiving node start exchanging data, while other neighbor nodes into sleep mode to save energy. Collision avoidance is not necessary in this scenario.

2.2.2.2 TRaffic-Adaptive Medium Access protocol (TRAMA)

TRAMA [169] could be categorized as a Time Division Multiple Access (TDMA) protocol. This protocol attains energy efficiency by avoiding collisions and switching to an idle state when there are no transmissions. To avoid collisions, TRAMA adapts its transmission schedule according to traffic information patterns. TRAMA assumes a single, time-slotted channel for data and control signal transmissions. Time is divided into sections of random access and scheduled access periods. TRAMA supports unicast, multicast, and broadcast traffic. TRAMA consist of three components: i) Neighbor Protocol (NP), ii) Schedule Exchange Protocol (SEP), and iii) Adaptive Election Algorithm (AEA).

In TRAMA, nodes start in random access mode where each node transmits at random slots. Nodes can join the network at random access periods. During this period, NP sends out small signaling packets to gather neighbor updates. If there are no updates, signaling packets are sent as keep-alive beacons. The signaling packets are used to maintain connectivity between neighbors.

The second component, SEP, sets up the traffic-based schedule. The schedule captures the traffic window for which the node can transmit. During scheduled access period, the node periodically broadcasts its schedule information to its one-hop neighbors. The schedules are sent along with every data packet. There is a special time slot called
2.2 Communication protocols and standardizations

ChangeOver which is the last slot in the current schedule interval. All nodes have to listen during the ChangeOver slot to synchronize their schedule.

The last component, AEA, determines the state of a node (transmit, receive, or sleep) based on the schedule information from its neighbors. For energy efficiency, nodes are switched to sleep most of the time. TRAMA guarantees delivery and energy efficiency with the expense of packet delays. It achieves high throughput and avoids collision.

2.2.2.3 Berkeley Media Access Control (B-MAC)

Unlike TRAMA, B-MAC [161] is a reconfigurable CSMA protocol that achieves low power processing, collision avoidance, and high channel utilization. B-MAC optimizes system performance by employing an adaptive preamble sampling scheme. A set of adaptive bi-directional interfaces is used to reconfigure the protocol based on the network load. B-MAC contains the following functionality: Clear Channel Assessment (CCA) and packet backoff, link-layer acknowledgement, and Low Power Listening (LPL).

For collision avoidance, B-MAC utilizes CCA to determine if the channel is clear. The basic assumption underlying CCA is that: i) a packet being transmitted will carry a signal intensity (called a Received Signal Strength Indication or RSSI) high enough to exceed a specified threshold; ii) all extraneous noise will fall below this threshold and be ignored. If the channel detects an RSSI value above the threshold it assumes the channel is in use by genuine traffic and will postpone packet transmission using packet backoff. The backoff time is either initially defined or randomly chosen.

B-MAC supports link-layer acknowledgement for unicast packets. When the receiver receives a packet, an acknowledgement packet is sent to the sender. To reduce power consumption, B-MAC employs an adaptive preamble sampling scheme called LPL. LPL performs periodic channel sampling by cycling through working and sleep periods. In the working period, the node’s radio is turned on to check for activities in the channel using CCA. If activities are detected, it will remain awake to receive the incoming packet. Once it receives the packet, it will go back to sleep. Idle listening occurs when the node is awake but there is no activity in the channel. In this case, a timeout will force the node to go back to sleep. All B-MAC functionality such as acknowledgements, CCA, and backoff can be changed through a set of adaptive bi-directional interfaces. By enabling or disabling B-MAC functionality, the throughput and energy consumption of a node can be customized.
2.2.2.4 Hybrid MAC protocol (Z-MAC)

In comparison to B-MAC, Z-MAC [93] is a hybrid MAC protocol that combines the strength of the TDMA and CSMA. Z-MAC achieves high channel utilization and low latency under high contention. It is robust to dynamic topology changes and time synchronization failures which commonly occur in WSN. Z-MAC uses CSMA as the baseline MAC scheme and a TDMA schedule to enhance contention resolution. This design results in high initial overhead, but after a long period of network operation this overhead will be reduced. The protocol uses an efficient and scalable channel scheduling algorithm for slot assignment and channel reuse. Unlike a normal TDMA protocol, in Z-MAC a node may transmit during any timeslot. It will always perform carrier sensing and transmit a packet when the channel is clear. This behavior will allow the reuse of a slot when the slot owner is not transmitting data. By mixing CSMA and TDMA, Z-MAC becomes more robust to timing failures, time-varying channel conditions, slot-assignment failures, and topology changes. Performance results show that Z-MAC is better than B-MAC under medium to high contention. Under low contention, B-MAC is slightly better in terms of energy.

2.2.2.5 Low power reservation-based MAC protocol (LPR-MAC)

The low power reservation-based MAC protocol [133] presents a solution to the problem of traffic adaptation while conserving the energy. To address the issue of energy conservation, this protocol employs both a clustered hierarchical network and a TDMA-based frame structure. In a clustered hierarchy network, nodes organize themselves into clusters and elect a cluster head for each cluster which is responsible for synchronizing all the nodes in the cluster to a TDMA schedule. The TDMA-like frame structure has contention-based slot reservation, schedule establishment, and slotted data transmission. Unlike traditional TDMA with fixed frame size, the protocol adapts the TDMA frame size according to the probability of successful data transmission. In this protocol, the cluster head increases the frame size if the number of failures exceeds a pre-defined value. On the contrary, if the number of failure is small, the cluster head will decrease the frame size. By adapting the frame size, the probability of success of packet transmission is increased. Nodes are able to effectively transmit at a higher data rate as a result of increasing throughput. In terms of wireless transmission, adaptive frame size shows significant energy savings due to less collisions and a higher probability of success.
2.2 Communication protocols and standardizations

2.2.2.6 Low power distributed MAC protocol (LPD-MAC)

The author of [33] presents a low power distributed MAC protocol which combines CSMA and multi-channel spread spectrum techniques. For a given frequency band, this band is partition into multiple channels. A channel and code is assigned to each node in the network. In order to avoid collisions, the assigned channel and code must be unique across each node’s two-hop neighbors. Two radios are used in this protocol, a low power wakeup radio and a normal data radio. The low power radio monitors the network and triggers the normal radio to wake up when there is data to transmit or receive. The normal radio switches between active and sleep modes. Results show that energy consumption for channel monitoring is almost negligible and average energy consumption is significantly reduced.

2.2.2.7 Spatial correlation-based collaborative MAC (CC-MAC)

CC-MAC protocol [202] exploits the spatial correlation of the data at the MAC layer to regulate and prevent redundant transmissions. The basis assumption under CC-MAC is that sensor nodes located near each other generate correlated measurements. To save energy, this protocol reduces the message volume need to be transmitted by filtering the measurements from highly correlated sensor nodes. CC-MAC has two components: event MAC (E-MAC) and network MAC (N-MAC). E-MAC filters out the correlated data packets while N-MAC prioritizes the routing packets. In order to recognize correlated event information, E-MAC protocol forms correlation regions, in which, a single representative sensor node is selected to transmit its data while all other sensor nodes are waiting for a specified period. At the end of each period, all nodes in that region except the representative sensor node go into a contention phase to choose a new representative. With E-MAC protocol filtering out correlated packets, the N-MAC protocol routes the packet to the sink using a priority-based method. The route-through packets are given higher priority over newly generated packets. Routing nodes use a backoff procedure to avoid collisions between multiple route-through packets transmitting at the same time. In terms of performance, CC-MAC protocol shows significant savings in energy, latency, and packet drop rate.

Table A.3 (Appendix A) compares the MAC protocols reviewed above.

2.2.3 Network layer

The network layer is responsible for routing of data across the network from the source to the destination. The challenges in the design of network protocols in a WSN are, besides power saving, robustness and scalability. The protocol should meet network resource constraints such as limited energy, communication bandwidth, memory, and
Chapter 2  Wireless sensor network

computation capabilities. Furthermore, the protocol should address issues of efficiency, fault tolerance, fair-ness, and security. A rich and diverse set of WSN routing protocols have been proposed. It is impractical to appraise each protocol as too many exist; a comprehensive survey can be found in [4, 8, 101]. Some representative examples from the major classes of WSN routing protocol are described below. Table A.4 (Appendix A) compares the reviewed protocols.

2.2.3.1 Packet routing by flooding

In classic flooding, a node wishing to send packet broadcasts to the shared wireless medium. Each neighbor receives a copy, and then rebroadcasts the packet to its neighbors unless it has previously broadcast the same packet. Flooding is utilized by many routing protocols in order to quickly maintain up-to-date routing information.

GOSSIP protocols [77] extend flooding by implementing probabilistic rebroadcast. Upon receiving a packet, each node independently decides whether it will be rebroadcast with probability \( p \), or dropped with probability \( 1 - p \). Clearly, the performance of GOSSIP depends on the choice of \( p \); however an appropriate gossip probability selection is generally difficult, and may need to vary across nodes and time.

Epidemic Routing Protocol (ERP) [198], is another approach based on flooding. In ERP, nodes have a bounded packet buffer; for a given packet, any node possessing a copy is a carrier. When a communication channel between any pair of nodes becomes available, they exchange one or more randomly selected packets, sometimes they may drop existing carried packets to make space. Eventually, the packet may reach its destination. ERP does not assume a connected path exists between packet source and sink, but it relies upon carriers coming into contact with another connected portion of the network through node mobility. However, packets may be lost if buffers are not sufficiently large to contain all packets in transit at any given time. This point makes ERP unrealistic for WSNs.

2.2.3.2 Table driven proactive and reactive routing

A proactive routing protocol calculates and maintains the network topology (frequently in a table form) before it begins the actual routing. Dynamic destination-Sequenced Distance Vector (DSDV) [159] is among the earliest and most widely referenced proactive routing protocols for ad-hoc networks. DSDV attempts to find the shortest path, defined as smallest number of hops, through a graph representing a network. Unlike many protocols, DSDV can work with data-link layer address (MAC address) or Network layer address (IP address). For WSNs with non-hierarchical address structures, this could be advantageous because routing could be tightly coupled with data-link layer
functions for efficiency. However, flat addressing schemes lead to increased overheads when nodes exchange routing tables.

Unlike the proactive protocol, a reactive routing protocol only searches for routing information if data packets need to be transmitted. Thus, the state exchange overhead is reduced to minimum. One of the most popular reactive routing protocol is Ad-hoc On demand Distance Vector (AODV) [158]. In AODV the routes from the sources to the destinations are created and maintained only when they are needed (on demand). The routing table stores the information about the next hop to the destination, as well as a sequence number which is received from the destination to indicate whether the information is up-to-date. The discovery of the route from source to destination is based on query and reply cycles and intermediate nodes maintain the routing table along the route. The advantages of AODV are loop-free, self-starting, and scalability.

2.2.3.3 Data-centric routing

Traditional network routing protocols data packets generated by lower network stack layers are wrapped as payload within higher layer packets. Under this model, network routers never consider packet payload in routing decision-making, passively forwarding data through the network unexamined and unmodified. In [105] the authors named this mechanism as address-centric routing in which the routing is done in an end-to-end manner, each source is treated independently.

Content-based routing [30] offers another approach to routing data within networks, in which, routing decisions are obtained by applying a set of rules to the data payload of each packet; no metadata is used. The ultimate destination of message packets is unknown and undefined, and the flow of packets is determined by the specific interests of the receiver rather than by an explicit destination address assigned by the sender.

There are some similarities between content-based routing and data-centric routing as described in [105]. The routes followed by packets are determined by packet content. An application-specific association exists between the producers of a given type of labeled data, and the consumers which express an interest in receiving data of this type. However, there are several important differences. In content-based routing approaches, each packet is routed individually toward potential consumers without attempting to aggregate multiple data streams derived from multiple sources. Whereas data-centric routing mechanisms generally connect multiple data sources to a single data sink.

Figure 2.3 is a simple illustration of the difference between three routing schemes. In the address-centric approach, each source sends its information separately to the sink. Two routes are: \((Source1 \rightarrow A \rightarrow Sink)\) and \((Source2 \rightarrow C \rightarrow B \rightarrow Sink)\). In the
content-based routing, the routes will be changed depend on the message’s content and the processing rule implemented on each intermediate node. For example, if node A discard all message, node B forward all message, node C forward message labeled “2” and discard message labeled “1”, then the routes will be: (Source1 \rightarrow B \rightarrow Sink) and (Source2 \rightarrow C \rightarrow B \rightarrow Sink). In the data centric-approach, the data from the two sources is aggregated at node B, and the combined data (labeled “1+2”) is sent from B to the sink.

Directed diffusion [95] is a popular data-centric routing protocol which combine the data coming from different sources en route (using in-network aggregation). In directed diffusion, data is named using attribute-value pairs. Such data can be event messages, which contain sensing data, or an interest message which describes a task required to be done by the network (i.e., a query for data). A sink may query for data by disseminating interests and having intermediate nodes propagate these interests. A flooding or gossiping routing protocol may be used in the interest propagation process. Each sensor that receives the interest sets up a gradient to indicate the sensor node from which it received the interest. This gradient field is used to form reverse paths towards the sink. This process continues until all gradients from the sources to the sink are set up. During this process, sensor nodes may use several local rules to define different gradient setup techniques. When the sources have data matching the interest, paths of information flow are formed from multiple paths, and then the best paths are reinforced by the sink to prevent further flooding. In order to reduce communication costs, data is aggregated on the way from source nodes to the sink. In a WSN using

Figure 2.3 The difference between three routing schemes (modify from [105])
Communication protocols and standardizations

directed diffusion, all nodes are application-aware. According to the application requirements a set of local rules are specified, that enables the diffusion protocol to achieve energy savings by selecting empirically good paths, and by caching and processing data in the network.

Other examples of data-centric protocols are query-based routing (COUGAR) [220], rumor routing [25], Gradient-Based Routing (GBR) [177], and Geographical Hash Table (GHT) routing [171].

2.2.3.4 Hierarchical / Geographical (location-based) routing

In hierarchical routing protocols, nodes play different roles in the network. Sensor nodes are clustered and the one with the higher residual energy is usually chosen as the cluster head. These cluster heads have some responsibilities like collecting and aggregating the data from their respective clusters and transmitting the aggregated data to the base station, when the rest of the nodes perform normal sensing task. A hierarchical routing protocol reduces the energy consumption within a cluster by performing data aggregation and fusion to decrease the number of transmitted messages. Furthermore, it allows the network to cope with additional load and to cover a large area of interest without degrading the service. Therefore, hierarchical routing protocols contribute to overall system scalability, lifetime, and energy efficiency of the network. However, most of these protocols have problems such as network partitioning (i.e., the elected cluster head has no other cluster head in its communication range). Furthermore, they are not capable of handling node mobility, and are hard to support time-critical applications due to the continuously cluster head evaluation procedure. The examples of hierarchical routing protocol are the Low-Energy Adaptive Clustering Hierarchy (LEACH) [82], Power-Efficient GAttering in Sensor Information Systems (PEGASIS) [117], and Threshold-sensitive Energy Efficient sensor Network protocol (TEEN) [12].

In geographical (or location-based) routing protocols, each node knows its own and its network neighbor’s positions, which is available using a small low-power GPS receiver. The position of the source and the destination node is encoded in the routing message to support selecting the energy efficient routing paths. The location information is used to calculate the distance between two particular nodes so that energy consumption can be estimated. A greedy forwarding mechanism is applied to forward a packet from the source to the destination following the shortest path (in terms of geographical distance). Another forwarding strategy is reputation-based forwarding using the packet reception rate of node’s neighbors. Four reputation-based forwarding schemes are proposed:
• Absolute reputation-based blacklisting: Each node black-lists all neighbors that have a reception rate below a certain threshold. Only neighbors closest to the destination with a reception rate above the threshold will receive the packet for forwarding.

• Relative reputation-based blacklisting: A node blacklists a different set of neighbors for each new destination. Blacklisting of neighbors depend on the node’s ranking within a set of neighbors. A node’s ranking depends on its distance to the destination and the reception rate. Relative reception-based blacklisting prevents all neighbors to be blacklisted as in absolute reputation-based blacklisting.

• Best reception neighbor: Best reception neighbor forwards packets to neighbors with the highest reception rate from the neighbors that are closer to the destination.

• Best reception rate and distance: Best reception rate and distance is based on the product of the reception rate and the distance. The node computes this product value for all neighbors that are close to the destination. The neighbor with the highest product value will be chosen.

Recent works on geographical-based routing protocols include geographic adaptive fidelity (GAF) [216], Hybrid Energy-Efficient Routing Scheme (HERS) [217].

2.2.3.5 Real-time routing

WSNs are inherently real-time systems because of their necessary interaction with the physical environment through sensors and actuators. Although hard real-time guarantees are difficult to achieve in the context of WSN due to unpredictable wireless communication environments and unreliable hardware platforms, soft real-time guarantees are achievable. In [194] the authors believed that the end-to-end deadline miss ratio is the most important metric for WSNs with soft real-time requirements, and that routing protocols using only local network information without flooding perform better in this metric because networks will become less congested without flooding.

The SPEED protocol [194] provides per-hop delay guarantees by applying a distributed feedback control scheme within a geographic routing strategy. It supports three types of real-time communication services; real-time unicast, real-time area-multicast and real-time area-anycast. End-to-end delay is proportional to the source-destination physical distance. Each node requires knowledge of only immediate neighbors, which leads to a minimal storage overhead. SPEED uses a stateless non-deterministic geographic forwarding algorithm. Relay nodes are selected by a probabilistic scheme, using only
2.2 Communication protocols and standardizations

local information. Packets are dropped only when no down-stream node can support the single-hop delay guarantee.

2.2.3.6 Secure routing

Sensitive WSN applications may have specific security requirements which need to be addressed. In that case, a security routing protocol is required to guarantee secure delivery of the data from the source to the destination. SecRout [99] is an example of this class. SecRout employs a two-level cluster-based approach to secure the network. The lower level contains sensors or cluster members while the upper level contains cluster heads. For secure packet delivery, SecRout uses symmetric cryptography to secure packets along the path. Each sensor node is given a unique identity (ID) and a preloaded key (KEY). The ID identifies the node and the KEY is used to secure messages sent to the sink. The sink is assumed to be a high power node with high memory and computation capability. The sink also knows about network topology and all sensor node information. A table containing each node’s ID and KEY pair is maintained by the sink. It is assumed that the sink cannot be compromised and can be trusted.

Secure data transfer starts with a sensor node encrypting its data packet using a cluster key. The cluster key is generated by the cluster head during the self-organizing phase and is shared among sensor nodes within the cluster. Upon receiving the encrypted data, the cluster head verifies the data using its cluster key. If the verification succeeds, the cluster head will decrypt the data. The cluster head collects data from all its members and then aggregates the data to form a new data packet. The new data packet will be encrypted with the cluster head’s pre-loaded key and sent to the sink via multi-hop routing. The sink receiving the packet again verifies the authenticity of the packet. If verification succeeds, it will decrypt the packet and store the information. SecRout guarantees that packets will reach the sink even if malicious nodes exist in the route. Routing packets and data packets contain only partial path information such as the next-hop neighbor. Each sensor node maintains a routing table containing partial routing path (previous and next node) to the sink. When a node is compromised, it will not be able to obtain information about the traversed intermediate nodes. SecRout provides route maintenance to update the routing table and trigger new route discovery when it detects a malicious node.

2.2.4 Transport layer

The transport layer ensures the reliability and quality of data transported from the source to the sink. The development of transport layer protocols in WSNs should be generic and independent of the applications in order to support multiple applications,
variable reliability, packet-loss recovery, and congestion control mechanism. Packet loss is an important issue that should be addressed in the transport layer, because any packet loss can result in wasted energy and degraded quality of service (QoS) in data delivery. Packet loss may be due to bad radio communication, congestion, packet collision, full memory capacity, and node failures.

In general, there are two approaches for packet recovery: hop-by-hop retransmission and end-to-end retransmission. In the first approach, the intermediate nodes cache the packet information in their memory. When an intermediate node detects packet loss, it will try to recover the loss by requesting retransmission to the previous relay node. This method is more energy efficient since the retransmission distance is shorter. In the end-to-end retransmission, the source caches all the packet information and performs retransmission when there is a packet loss. End-to-end retransmission allows for variable reliability, whereas hop-by-hop retransmission performs better when reliability requirements are high.

A congestion control mechanism monitors and detects congestion. In order to save energy, before congestion occurs, the source is notified to reduce its sending rate. Congestion control helps reduce retransmission and prevents sensor node buffer overflow. As in packet-loss recovery, there are two approaches to congestion control: hop-by-hop and end-to-end. The hop-by-hop mechanism requires every node along the path to monitor buffer overflows. When congestion is detected by a sensor node, all nodes along the path change their behavior. The end-to-end mechanism relies on the end nodes to detect congestion which are flagged when timeout or redundant acknowledgements are received.

There are tradeoffs between hop-by-hop and end-to-end approaches for packet-loss recovery and congestion control mechanism. Depending on the type, reliability, and time-sensitivity of the application, one approach may be better than the other. Below we review some existing transport layer protocols in WSNs in an attempt to address the above design issues. Table A.5 (Appendix A) compares the reviewed protocols.

2.2.4.1 Delay sensitive transport (DST)

DST protocol [76] tries to address the issue of congestion control and reliable packet delivery for WSN applications that involve even detection and tracking. The assumption DST makes is that, in this kind of application, sensors-to-sink transport does not require 100% reliability due to the correlation among sensor readings. Based on this assumption, DST attempts to guarantee the desired reliability level in the event-to-sink communications. The event-to-sink communication can be considered as the collective
communication from the sensors in the event area to the sink. In DST the following definitions are introduced:

- The observed delay-constrained event reliability ($DR_i$): is the number of packet received within a certain delay bound at the sink over a specified interval $i$.
- The desired delay-constrained event reliability ($DR^*$): is the minimum number of data packets required for the event to be a reliable detection.
- The delay-constrained reliability indicator ($\delta_i$) is the ratio of the observed and desired delay-constrained event reliabilities: $\delta_i = DR_i / DR^*$

DST has two components: a reliable event transport mechanism and a real-time event transport mechanism. Reliable event transport mechanism measures the delay-constrained reliability indicator ($\delta_i$) to determine if appropriate action is needed to ensure the desired reliability level for event-to-sink communication. If $\delta_i > 1$, the event is considered to be reliable. Otherwise, the report rate of the sensors must be increased to assure that the desired reliability level is met. DST also assures reliable and timely event detection within the event-to-sink delay bound.

The real-time event transport mechanism uses this event-to-sink delay bound delay to achieve the application specific objectives. The event-to-sink delay is a measure of the event transport delay and event process delay. Event transport delay is the time between the event occurring and when the sink receives it. Event process delay is the processing delay at the sink. For congestion detection, DST measures buffer overflow at each node and computes the average delay. Upon congestion, sensor nodes inform the sink of the congestion situation. The sink in response would adjust the reporting rate of the sensors. Simulation experiments show that DST achieves reliability and timely event detection with minimum energy consumption and latency [76].

2.2.4.2 Sensor transmission control protocol (STCP)

STCP [98] is a reliable transport layer protocol that provides variable reliability, congestion detection and avoidance, and support of multiple applications in the same network. Functionalities of STCP are executed at the base station. The base station is assumed to have high processing capability, storage, and power to communicate with all the nodes in the network.

In STCP, a source node must transmit a single session initiation packet, which contains information about the number of flows from the node, the type of data flow, transmission rate, and required reliability, to the base station before sending data. The sensor node must wait for an acknowledgement from the base station before transmitting data. For continuous data flows, the base station estimates the time of
arrival of each packet from each source. If a packet is not received by the base station within a given period of time, depending on a reliability factor the base station may ask the source node for retransmission. The reliability factor is a measure of the fraction of packets that are successfully received. If the reliability factor goes below the required level, the base station sends out a negative acknowledgement (NACK) to the source node for retransmission. In order to support retransmitting, each source node stores its transmitted packets in a buffer. The base station also sends out a positive acknowledgement (ACK) for each packet received from a source node. Based on these ACK, source nodes can compute the reliability of the packet reaching the base station. If the computed value is more than the required reliability, the node will not buffer the packet to save storage space.

To implement congestion control, a congestion bit is used. Every sensor node maintains two thresholds in its buffer: low and high thresholds. When the buffer reaches the lower threshold, the congestion bit is set with a certain probability. Once the buffer reaches the higher threshold, the congestion bit is set for all packets. The base station extracts the congestion bits from packets and determines whether to notify the source to reduce its transmission rate or re-route packets along a different path.

2.2.4.3 Pump slowly, fetch quickly (PSFQ)

PSFQ [204] is a transport protocol that increases the reliability and scalability by dividing transmission into a series of single hops. PSFQ operates in three functions: pump operation (message relaying), fetch operation (packet-loss recovery), and report operation (status reporting). The pump operation controls the rate at which data packets are passed along to the network. This operation is based on a simple scheduling scheme which uses two timers, $T_{\text{min}}$ and $T_{\text{max}}$. The source is scheduled to send packets to its neighbors every $T_{\text{min}}$. By waiting at least $T_{\text{min}}$, a node is given the opportunity to recover missing packets and reduce redundant broadcasts. In a relay node, the received packet will be delayed for a random period between $T_{\text{min}}$ and $T_{\text{max}}$, and then relayed to its neighbors. $T_{\text{max}}$ can be used as a loose upper delay bound for the last hop to successfully receive all packets.

The fetch operation is called when a relay node detects packet loss. This operation requests a retransmission of the lost packet from the neighboring nodes. Because the data loss is often correlated in time, PSFQ has the ability to aggregate loss such that the fetch operation attempts to batch up all message losses in a single operation.

Lastly, the report operation provides a feedback status report to the user. A status report message travels from the farthest target node in the network to the requesting
user. Along the path, each node appends its report message in an aggregated manner into the original message. Results show that PSFQ outperforms the several transport protocols in terms of tolerance, communication overhead, and delivery latency.

2.2.4.4 Price-oriented reliable transport protocol (PORT)

PORT [219] present an approach to minimize energy consumed by avoiding high end-to-end communication cost which is the measure of the amount of energy consumed to deliver a packet from the source to the sink. To achieve the necessary level of reliability and minimize energy, the source’s transmitting rate is dynamically adjusted to assure that the sink obtains enough information on the phenomenon of interest. The ideal behind PORT is that, when a phenomenon of interest occurs, nodes closer to the phenomenon will contain more information and less error. The sink adjusts the reporting rate of each source based on the information provided about the physical phenomenon.

Additional information, which is used in the adjustment, is the node price. Node price is the total number of transmission attempts made before a successful packet is delivered from the source to the sink. It is a metric used to evaluate the energy cost of the end-to-end communication from the source to the sink. When congestion occurs, communication cost increases with respect to packet loss. The sink uses the communication cost information to slow down the reporting rate of the appropriate source and increase the reporting rate of other sources that have lower communication cost since reliability must be maintained.

2.2.4.5 Congestion detection and avoidance (CODA)

CODA [205] is an energy-efficient congestion control scheme that can quickly mitigate congestion, once detected. CODA has three components: congestion detection, hop-by-hop backpressure, and multi-source regulation. CODA detects congestion by monitoring buffer occupancy and measuring channel load. Monitoring of the buffer size requires only a small amount of overhead and processing. When buffer occupancy is high, sensors listen to the local channel load conditions to detect congestion. Once congestion is detected, the sensor node broadcasts a suppression message to its neighbors and makes adjustment to prevent congestion downstream. It also broadcasts a backpressure message upstream to the source. Each upstream node receiving the backpressure message determines whether or not to propagate the message. Depending on the congestion policy, a node can prevent further congestion build up by dropping the incoming data packets or adjust their sending rate. In the event of a persistent congestion, CODA uses a closed-loop multi-source regulation method to assert
congestion control over multiple sources from the sink. When the source node event rate is less than some fraction of the maximum theoretical throughput of the channel, the source regulates its own rate. Upon exceeding this value, the source node is most likely to be contributing to congestion. In this case, the source enters sink regulation. The sink sends a message to the source at a pre-defined event rate that the sink computed. When congestion is relieved, the sensor node would then regulate itself again without the sink. Results show that CODA can improve performance and reduce energy usage.

2.2.4.6 GARUDA

GARUDA [181] is a reliable downstream data delivery transport protocol for WSNs. It addresses the problem of reliable data transfer from the sink to the sensors. Reliability is defined by four categories: i) guaranteed delivery to the entire field, ii) guaranteed delivery to a sub-region of sensors, iii) guaranteed delivery to a minimal set of sensors to cover the sensing region, and iv) guaranteed delivery to a probabilistic subset of sensors. GARUDA’s design is a loss-recovery core infrastructure and a two-stage NACK-based recovery process.

The core infrastructure is constructed using the first packet delivery method that guarantees first packet delivery using a Wait-for-First-Packet (WFP) pulse. WFP pulse is a small finite series of short duration pulses sent periodically by the sink. Sensor nodes within the transmission range of the sink will receive this pulse and wait for the transmission of the first packet. The first packet delivery determines the hop-count from the sink to the node. Nodes along the path can become candidates for the core. A core candidate elects itself to be a core node if it has not heard from neighboring core nodes. In this manner, all core nodes are elected in the network. An elected core node must then connect itself to at least one upstream core node. To avoid link under-utilization at low loads, GARUDA uses an out-of-order forwarding strategy which allows subsequent packet to be forwarded even when a packet is lost.

GARUDA uses a two-stage loss-recovery process. The first stage involves core nodes recovering the packet. When a core node receives an out-of-sequence packet, it sends a request to an upstream core node notifying that there are missing packets. The upstream core node receiving that message will respond with a unicast retransmission of the available requested packet. The second stage is the non-core recovery phase, which involves non-core nodes requesting retransmission from the core nodes. A non-core node listens on all retransmissions from its core node and waits for completion before sending its own retransmission request.
2.2 Communication protocols and standardizations

2.2.4.7 Event-to-sink reliable transport (ESRT)

ESRT protocol [3] is developed for reliable event detection with minimum energy expenditure. ESRT uses a congestion control mechanism to reduce energy consumption while maintaining the desired reliability level at the sink. ESRT algorithm is run mainly at the sink. The sink computes the reliability factor that is a measure of the data packets received from the source nodes to the sink, and reporting frequency at each interval. The computed reliability factor is compared against an application-defined desired reliability. If the computed reliability is greater than the desired reliability, ESRT would reduce the reporting frequency of the source nodes. If the computed reliability is lower than the desired reliability, ESRT would increase the reporting frequency of the source nodes to achieve the desired reliability. At each interval, the sink broadcasts the new reporting frequency to source nodes in the network. Upon receiving the information, source nodes adjust their reporting rate. In ESRT, the congestion control mechanism is based on monitoring the routing buffer of each sensor nodes. The congestion occurs when a sensor node’s buffer is overflow. Upon experiencing congestion, the sensor node sets the congestion notification bit flag in its outgoing packets. The sink receiving these packets along with the computed reliability factor determines the state of the network and acts accordingly. Simulation results show that ESRT is able to attain the desired reliability level with minimum energy expenditure under different network states with random and dynamic topologies.

2.2.5 Standardizations

As we discussed above, many communication protocols are developed for WSNs. However, most of these protocols use standard-based networking and RF solutions. The recent release of standards from the IEEE [88], the Internet Engineering Task Force (IETF) [90], and the International Society of Automation (ISA) [96], brought the technology out of research labs and developed the numerous commercial products. There have been many contributions to the standardized protocols for low-power devices such as ZigBee [9], IETF 6loWPAN [91], IETF routing over low power and lossy networks (ROLL) [92], WirelessHART [66], ISA SP-100 [96], IEEE 802.15.4 [89], and IEEE 802.11 [87]. In the following, we describe these standards in detail and highlight the key ideas.

2.2.5.1 IEEE 802.15.4

Many promising standards and commercial systems are based on the IEEE 802.15.4 standard. This standard is for a low data rate wireless network connecting personal devices with low-cost and efficient energy consumption. Such wireless network can be a
personal health monitor network, intra-vehicular wireless networks, home or smart office network. The IEEE 802.15.4 radio standard and the ZigBee emerge as the prevalent choice for industrial and smart building applications. There are also many discussions of IEEE 802.15.4 for the routing over low power and lossy networks in the IETF working group [92]. Recently, many task groups launched the IEEE 802.15.4 family for specific applications of WSNs. The task groups of IEEE 802.15.4 include:

- **IEEE 802.15.4a**: This amendment specifies two additional physical layers using Ultra-wideband (UWB) and Chirp Spread Spectrum (CSS) which can be used to provide communications and high precision ranging/location capabilities, high aggregate throughput, ultralow power, scalability to data rates, longer range, and lower cost.

- **IEEE 802.15.4e**: The intent of this amendment is to enhance and add functionality to the IEEE 802.15.4 MAC protocols that are required to enable support for various kind of applications including factory automation, process automation, asset tracking, general sensor control (industrial/commercial, including building automation), home medical health/monitor, telecom application, neighborhood area networks, and home audio. A multi-channel mechanism is also included.

- **IEEE 802.15.4f**: This amendment defines new wireless physical layer to support the active RFID system. The enhancements to the IEEE 802.15.4 standard MAC layer are also included.

- **IEEE 802.15.4g**: This amendment defines new wireless physical layer to provide a global standard that facilitates very large scale process control applications such as the smart-grid power network presented in [11].

### 2.2.5.2 ZigBee

The ZigBee [9] standard was published by the ZigBee Alliance. This standard defines a layered model of interworking protocols and interfaces for WSNs. In the ZigBee design, the lightweight resource demands, security, and low power consumption are the highest priority. The design goals also include scalability for systems of up to 65,000 nodes, ease of deployment, long battery life, robust security, and low cost. Communication in ZigBee networks is implemented by RF transmission using the license-free and globally available 2.4 GHz band. The IEEE 802.15.4 standards are used to define the lower layers of the ZigBee stack such as the MAC layer and the physical layer. Above IEEE 802.15.4, ZigBee adds a network layer, an application layer, an application frame-work, and a number of security services.
2.2 Communication protocols and standardizations

ZigBee allows to the implementation of various kinds of network topologies. In a star network, a ZigBee coordinator starts the network and all other network members are directly associated with the ZigBee coordinator. The ZigBee coordinator can be co-located with the personal area network (PAN) coordinator from the underlying IEEE 802.15.4 network. In a tree network, the ZigBee routers form a tree that is rooted at the ZigBee coordinator, whereas in mesh networks, the network topology might be a general mesh including many ZigBee routers and ZigBee coordinators.

As ZigBee is an open standard, it is possible for multiple manufacturers to supply compatible ZigBee devices. These devices can readily be composed into a functioning WSN, without any modification. This interoperability support is a significant factor in the popularity of ZigBee as WSN begin to find commercial applications beyond the research community.

2.2.5.3 WirelessHART

WirelessHART [66] is a promising solution for the replacement of the wired HART protocol in industrial contexts. However, power consumption is not a main concern in WirelessHART because the data link layer is based on TDMA, which requires time synchronization and pre-scheduled fixed length time-slots by a centralized network manager. Such a manager should update the schedule frequently to consider reliability and delay requirements and dynamic changes of the network, which requires complex hardware equipment. This is in contrast with the necessity of a simple protocol that is able to support the devices with limited energy and computing resources.

2.2.5.4 ISA SP-100

ISA SP-100 [96] is currently working on a series of standards addressing the adoption of wireless technologies in different industries. ISA-SP100.11a standard addresses non-critical process applications that can even tolerate delays up to 100ms. ISA SP-100 supports the IEEE 802.15.4 standard, and inherits some of its properties: low-rates (up to 250kbit/s), and easy implementation on end devices. In addition, a data-link layer and an adaptation layer below MAC layer is introduced. The data-link layer controls the frequency hopping and adds a TDMA scheme. Similar to ZigBee, ISA SP-100 standard also targets overlapping application areas, and uses the IEEE 802.15.4 standard for the underlying wireless technology.

2.2.5.5 IEEE 802.11

The IEEE 802.11 [87] family of WLAN standards is composed of a number of specifications that primarily define the physical and MAC layers of the realm of WLAN
technologies. This standard has also been considered extensively in the context of wireless industrial communications [138, 180, 188]. Similar to other standards from the IEEE 802.x series, the IEEE 802.11 MAC suggest the IEEE 802.2 logical link control (LLC) [86] as a standard interface to higher layers. Because IEEE 802.11 is a WLAN standard, its key intentions are to provide high throughput and a continuous network connection rather than energy efficiency.

2.2.5.6 Routing Over Low power and Lossy networks (ROLL)

ROLL [92] is focused on routing issues for low power and lossy networks (LLNs). LLNs are made up of many embedded devices with limited power, memory, and processing resources. They are interconnected by a variety of links, such as IEEE 802.15.4, Bluetooth, low power WiFi, and wired or other power-line communication (PLC) links. LLNs are transitioning to an end-to-end IP-based solution to avoid the problem of non-interoperable networks interconnected by protocol translation gateways and proxies. The working group focuses on routing solutions for a subset of these: industrial, connected home, building and urban sensor net-works for which routing requirements have been specified. These application specific routing requirements will be used for protocol design. The framework will take into consideration various aspects including high reliability in the presence of time varying loss characteristics and connectivity while permitting low power operation with very modest memory and CPU pressure in networks potentially comprising a very large number of nodes.

2.2.5.7 IPv6 over Low power Wireless Personal Area Networks (6LoWPAN)

The 6LoWPAN [91] standard is concerned with transmitting IPv6 packets over IEEE 802.15.4 networks, enabling sensor nodes to take part in the Internet. The 6LoWPAN protocol stack consist of the 802.15.4 physical layer and MAC layer, adaptation layer, network layer, transport layer, and application layer. In theory, 6loWPAN has one capability that distinguishes itself from all the above standards for WSNs; it is compatible at the network layer with normal internet capable devices. As this standard is still under development, many contributions have been made at these layers, especially at the adaptation layer, network layer and transport layer.

Because IPv6 packet sizes are larger than the frame size of IEEE 802.15.4, an adaptation layer is used. The adaptation layer is responsible for header compression. With header compression, smaller packets are created to fit an IEEE 802.15.4 frame size. Until now, several open 6LoWPAN stacks have been available; however, the scalability of LoWPANs are still needed because WSN nodes have limited resources and most WSNs adopt sleep & wakeup running modes to increase network lifetime.
2.3 Operating system

Operating systems (OSs) specifically designed taking into account the requirements and constraints of sensor nodes, manage the complexity of physical components, and provide the necessary abstractions to programmers. In general, OSs cover the required functionality: network protocols (i.e., MAC, routing, discovery, synchronization), local timing, sensing, data storing, power-efficient modes, and so on. Currently, there are several popular WSN Oss [61] such as TinyOS, Contiki, Mantis and LiteOS. Table A.6 (Appendix A) compares these operating systems in detail.

Based on the execution model, most OSs designed for sensor nodes can be grouped into one of these two categories: event-based or thread-based, which are discussed in the next section.

2.3.1 Event-based and thread-based model

The event-based model consists of a program which is implemented as a set of independent functions or event handlers. Every event handler is triggered as a response to one event which could be internal or external. When an event happens, the event handler is executed atomically to completion then returns to the caller. In this semantic, there is no possibility of blocking situations while running, and therefore it eliminates the overhead imposed by the context switching. During its execution, every event handler uses a single memory space allocated for this purpose by the operating system. Due to the fact that the stack is shared, the event-based model reduces the amount of used memory.

In a thread-based system, the set of actions is performed by execution entities called threads, which requires its own stack to manage state information. Therefore, context switching and race conditions could happen while a thread is running, which means that inter-thread synchronization and scheduling mechanisms must be provided by the OS. In addition, because the threads occupy resources while running, there may not be sufficient resources to divide among a large number of threads. Therefore, some OSs support only a limited number of threads and keep them in a pool. Each thread in the pool waits for a task assignment. Once a request is received, it is assigned to an available thread in the pool. After completion of the task, the thread returns to the pool and awaits the next assignment. If all the threads in the pool are used, the upcoming request will be held in a queue until the next thread returns to the pool. In this way the OS can keep the number of threads to a manageable size. Figure 2.4 illustrates the stack usage in event-based and thread-based approaches.
Compared to thread-based model, event-based model is more resource effective; however, in terms of programming style the event-based model has considerable disadvantages. In general, an event-driven program often employs a complex state machine, in which the code is unstructured and non-linear, the control flow is not explicit and the execution order is undefined. As a result, event-driven code is hard to read and debug, and the development tools often lack mechanisms to explicitly express the program structure. On the contrary, the thread-based programming style is more natural for programmers, since the writing is sequential. Figure 2.5 compares the control flow in both models, and Table 2.1 summarizes the main features of the two paradigms.
2.3 Operating system

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Event-based model</th>
<th>Thread-based model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic</td>
<td>Run-to-completion</td>
<td>Preemptive</td>
</tr>
<tr>
<td>Resource</td>
<td>Shared stack</td>
<td>One stack per thread</td>
</tr>
<tr>
<td>Control flow</td>
<td>State machine</td>
<td>Linear</td>
</tr>
<tr>
<td>Programming flow</td>
<td>Unstructured</td>
<td>Sequential</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Asynchronous</td>
<td>Blocking</td>
</tr>
<tr>
<td>Concurrency model</td>
<td>Non explicit by event-handlers</td>
<td>Explicitly by threads</td>
</tr>
</tbody>
</table>

Table 2.1 Main characteristics of event-based and thread-based model

2.3.2 Tiny OS

TinyOS [115] was the first open source OS specifically designed for WSN motes. TinyOS can support concurrent programs with very low memory requirements. The OS has a footprint that fits in 400 bytes. TinyOS component library includes network protocols, distributed services, sensor drivers, and data acquisition tools. TinyOS is written in nesC [113], a component-based programming language based on C that allows programming interfaces and components. nesC distinguishes two kind of components: the first one is implementation component (or module), which specifies what interfaces use and provide the component, and contain its implementation. The second one is specification component (or configuration), which describes the relations among interfaces used and provided by components.

This component model is one of the strong points of TinyOS, because it supports modularity programming. TinyOS also provides flexible abstractions for the different levels including hardware level and OS abstraction level. In TinyOS, components have three computational abstractions: commands, events, and tasks. Commands and events are used for inter-component communication, whereas tasks are used inside a component to perform general-purpose background processing. A command is a request to perform some service, while the event signals the completion of the service, and a task is a function which a component tells TinyOS to run later, rather than now. To manage tasks, TinyOS uses a task queue which is processed in First In First Out (FIFO) order by a scheduler. The scheduler goes into a loop waiting for incoming events or tasks. The events have higher priority than the tasks. If there is a pending event, the schedule first executes the associated handler event. On the contrary, if there are no pending events, the scheduler executes the waiting task in the queue. When a task is executed, it runs to completion before the next task is run. But if an event arrives when one task is being executed, the scheduler interrupts this task, saves its state and executes the associated handler event, and finally restores the execution of the task. Figure 2.6 shows the TinyOS architecture.
The communication model used in TinyOS is based on an Active Message component which multiplexes the physical radio between different communication channels. In TinyOS every message incorporates a unique identifier, which is related to the message handler to be invoked when the message arrives at the target node. Message handlers run to completion and its execution is very fast; therefore, it reduces the latency in communications. In this way, Active Messages paradigm fits very well with the asynchronous and event-based execution model of TinyOS.

The second version of TinyOS was released in November of 2006. Besides supporting new hardware platforms, it incorporated multiple improvements over the earlier versions, among which the following are remarkable:

a) Hardware access is organized in a Hardware Abstraction Architecture (HAA) of three levels:
   - The Hardware Independent Layer (HIL) is the top layer in the architecture, which is independent of the underlying hardware.
   - The Hardware Adaptation Layer (HAL) provides high-level abstractions of the underlying hardware; therefore, it is platform-specific.
   - The Hardware Presentation Layer (HPL) is the lowest level of the architecture, and it abstracts away the raw hardware as nesC interfaces.
2.3 Operating system

b) The scheduler manage is improved to support more posted tasks. In TinyOS 2.x, a post of a task will only fail if and only if the task has been previously posted and its execution has not started. This semantic is achieved by locating one byte of state per task, to indicate if a task needs to be executed again.

c) The boot sequence and initialization is split into two interfaces: Init and StdControl, which hid the initialization of all internal components related to hardware and operating system.

d) Efficient power management is another advantage of TinyOS 2.x. It takes into account the specific hardware (microcontroller and devices) for energy saving.

e) New interfaces, components and data definition are incorporated into the Active Message mechanism. New network protocols such as dissemination and collection are added. The network model in TinyOS 2.x gives support to 6LoWPAN, but due to the limited resource of sensor nodes, the TinyOS implementations do not fully incorporate the requirements for a full IPv6 stack.

2.3.3 Contiki

Contiki [52] is a lightweight operating system designed for sensor motes, developed in Europe in the Swedish Institute of Computer Science by Adam Dunkels. This OS is written in the C programming language. A Contiki program has a bigger footprint than a TinyOS program, a typical Contiki application consumes 2 kilobytes of RAM and 40 kilobytes of ROM because it incorporates new features like: multitasking kernel, preemptive multithreading, proto-threads, TCP/IP networking. Contiki could be considered the second most extended operating system for programming sensor nodes because it presents some contributions with respect to TinyOS:

a) Contiki supports load dynamic of programs on the top of the operating system kernel; therefore, applications can be more easily updated. In Contiki, the update code/program for motes can be directly downloaded from the sensor network. This feature is one of the main advantages of Contiki, whereas other operating systems frequently generate only one image of the system including both application and operating system kernel.

b) Unlike most WSN operating systems, which use an event-driven programming model in order to reduce the overhead of the system, Contiki uses a very lightweight thread-based mechanism, called proto-thread [54]. The main features of proto-threads are: very small memory overhead (only two bytes per proto-thread), no extra stack for a thread, and highly portable.

c) Contiki also supports a preemptive multi-thread library in the top of an event-driven kernel. This library can be included on demand by the application.
Figure 2.7 shows the block diagram of the Contiki OS architecture. The Contiki execution model combines events and threads into proto-threads which provide a thread-like programming style (blocking model and sequential flow). Proto-threads provide a set of high-level programming abstractions, which are implemented as C preprocessor macros. Every proto-thread consists of a C function and a local continuation. The local continuation is a two byte pointer representing the address of the C function implemented by the proto-thread. The local continuation is completely updated through these C macros, which basically perform two internal operations: set or resume the state of the C function (all CPU registers including the program counter but excluding the stack). In this way, the proto-threads can provide conditional blocking using a shared-stack.

A process in Contiki is a single proto-thread, which implements a C function. An application can include several proto-threads, which will be executed in the order decided by a scheduler. In addition, a process (or proto-thread) can also be executed automatically when the application starts. The scheduler takes each proto-thread from a FIFO queue and executes it, until there is a blocking condition. In the case that there is no blocking condition, every proto-thread runs to completion without being interrupted by other proto-threads.
Contiki supports a rich set of communication stacks including both IPv4 and IPv6. uIP [51] is a lightweight version of TCP/IPv4 stack for 8-bit microcontrollers. The basic idea is to remove from the full TCP/IP stack services rarely used by sensor node: it supports only a single network interface and contains TCP, UDP, IP and ARP protocols. In this way, sensor nodes can partially communicate to other hosts in the Internet. Rime [53] is another lightweight protocol stack which intends to offer a set of communication protocols arranged in a layered fashion. Interaction among Rime protocols is performed via callbacks, which means that a certain event is signaled to the upper layer (just as TinyOS events). Nodes using Rime must agree on the channel to use: every channel is identified by a 16-bit number. Figure 2.8 depicts the Rime stack protocols suite. Rime consists of 16 protocols at different layers, every one of them implementing different routing protocols:

- **abc**: Anonymous Best-Effort Single-hop Broadcast, is the most basic protocol. It broadcasts packet to all neighbors in the channel. There is no information about the packet source.
- **sabc**: Stubborn Best-Effort Single-hop Broadcast provides stubborn anonymous best-effort single-hop broadcast.
- **trickle**: Reliable multi-hop flooding. It sends a single packet to all nodes on the network.

![Figure 2.8 Rime protocol stack](image)

Figure 2.8 Rime protocol stack
• uabc: Unique Anonymous Best-effort Single-hop Broadcast. It broadcasts packet to all neighbors within one time interval. There is no information about the packet source.
• ibc: Identified Best-Effort Single-hop Broadcast. It broadcasts a packet to all neighbors. The source address is added into the outgoing packets.
• uc: Best-Effort Single-hop Unicast. It sends a packet to a single destination in a single-hop.
• uibc: Unique Identified Best-effort Single-hop Broadcast. It broadcasts packet to all neighbors within one time interval. The source address is added into the outgoing packets.
• rudolph1: Multi-hop Reliable Bulk Transfer. This protocol implements a multi-hop reliable bulk data transfer mechanism.
• rudolph0: Single-hop Reliable Bulk Transfer. This protocol implements a single-hop reliable bulk data transfer mechanism.
• mh: Best-Effort Multi-hop Unicast. It sends a packet to a single destination, using multi-hop forwarding.
• suc: Stubborn Single-hop Unicast. This protocol repeatedly sends a packet to a single-hop neighbor using the unicast primitive (uc).
• nf: Best-Effort Multi-hop Flooding. It sends a packet to all nodes in the network.
• ruc: Reliable Single-hop Unicast. Reliable single-hop sending of packets to a single destination. It uses ACKs and retransmissions to ensure that the destination has received the packet.
• tree (also known as collect): Hop-by-hop Reliable Data Collection Tree Routing, implements a hop-by-hop reliable data collection mechanism.
• route-discovery: Best-effort Route Discovery does route discovery for Rime protocols.
• mesh: Hop-by-hop Reliable Mesh Routing, sends packets using multi-hop routing to a specified receiver somewhere in the network.

2.3.4 Other OSs for motes

Mantis Operating System (MOS) [21] is a lightweight preemptive multi-threading operating system for WSN motes. MOS is written in C and it supports application development in C using a POSIX-like programming style. This is one of the strong points of this operating system. In order to support multi-threading, MOS allocates stack space for every thread (the size of the stack is configurable and ranges between 128 bytes and the maximum available RAM). In MOS, threads are managed by a priority-based scheduler using a thread table to store the thread’s state such as current stack pointer,
stack boundary information, pointer to thread starting function, thread priority level, and pointer to next thread. The scheduler executes the threads according to their priority from the highest to the lowest. If a new thread with higher priority than the thread currently executing is scheduled, the current thread will be preempted by the new one. When there is no thread in the queue, the system goes to sleep mode.

LiteOS [28] is a UNIX-like, multithreaded operating system with object-oriented programming support for wireless sensor networks. It includes several features of the Unix systems such as a thread-based programming mode, a hierarchical file system, a shell or the programming environment. LiteOS follows a modular architecture design which composed of three main components: LiteShell, LiteFS, and the Kernel.
Chapter 3  Model-driven software development

3  Model-driven software development

3.1  Concept and definition

The ideas of models, modeling, and model transformation are the basis of the model-driven development (MDD) approach [71, 128, 184, 211] – a software production paradigm that is gaining acceptance in the last years. MDD approach suggests that developer should first develop a model of the system under study, which is then transformed into the real thing (i.e., an executable software entity). In the context of MDD, a model, which is defined as “an abstraction of a (real or language-based) system allowing predictions or inferences to be made” [106], has the following important aspects:

- mapping: based on a real entity
- abstraction: only describe important details, remove everything else
- isomorphism: conclusions hold for the real world
- pragmatics: the model is usable in place of the original for some purpose

Among the MDD approaches, the Model-Driven Architecture (MDA) developed by the Object Management Group (OMG) [127, 145] is the most promising and widely known one which has been used as the basis for many other later MDD methods. It is based on a set of emerging standards for how to define a set of models, notations, and transformation rules. MDA defines three viewpoints (or levels of abstraction) for analyzing systems. For each viewpoint, we can define a model to represent the system under study. The three models are:

- the computational-independent model (CIM)
- the platform-independent model (PIM)
- the platform-specific model (PSM)

A CIM is a domain or business model which represents domain specific information, without the details of the structure of the system or the implementation technologies. The PIM is a computation-dependent model, which describes the system’s services and functions such as transactions, events and security, but it does not consider the characteristics of specific computer platforms. In other words, the PIM is a model assumed to be executed on a technologically independent virtual machine. A PSM combines the specification of a PIM with the platform specific specification. A platform specific specification is supported by a particular platform model, such as a CORBA or Java component model.
Three models above are defined using metamodels, which are another key concept used in MDA. A metamodel is defined as “the language definition of a model that describes which parts a model consists of and is used to build valid models” [13, 106]. Since a metamodel itself is a model, it can be represented in a modeling language. In MDA, there is a specialized modeling language for defining metamodels, and that language is defined in the metametamodeling layer of a specific modeling architecture (see Figure 3.1). The OMG defines four layers of the modeling architecture including:

- **MOF**: The Meta Object Facility (MOF) is a metametamodel and describes how to specify metamodels. MOF also describes itself, and it is therefore also an instance of itself. An example of MOF is Ecore models [186].
- **DSL**: A Domain Specific Language is an instance of the MOF and is built to describe instances of the modeled domain.
- **User Model**: The model which is developed by user to describe a concrete problem and its solution in the specific domain modeled by the DSL.
- **Instance**: The instance is the lowest level of the modeling hierarchy. It can be executed or transformed into an instance of a target model.

![Figure 3.1 A sample of model-to-model transformation](image)

Model-to-model transformation is the last part of MDA, which is described in this session. A model-to-model transformation is defined as “the transformation rules that describe how to automatically generate target models from source models” [131]. The source and target models in the definition can each also be a single model. The most importance for a model-to-model transformation is that it is a computable function. Figure 3.1 depicts the relationship between the modeling architecture and a model-to-model transformation to the C programming language.
3.2 Domain Specific Language

In the context of MDD, domain-specific languages [41, 124] can be used as modeling languages for defining software at higher level of abstractions. It offers an appropriate suite of notations and abstractions for expressing the application domain’s concept, and it is usually more declarative than imperative. The person supposed to use a DSL is the domain expert rather than the skilled programmer. With DSLs, domain experts can concentrate on what to compute, and the programmer could rather focus on how to compute. In this way, a DSL mediates the collaboration between domain experts and programmers, leading to software that is more usable, and more reliable.

DSLs are used in a broader context of software engineering, and the benefits of DSL have been delivered in various domains [41, 100, 109, 124, 151, 190]. Classical and well-known examples include SQL for database queries, VHDL for hardware design, Matlab for technical computing, and HTML for web content. Of course there are also a number of disadvantages of using a DSL. Firstly, designing, implementing and maintaining a DSL is a rather elaborate job and this involves a lot of costs. Typical standard tasks for creating any new programming language are also required for DSL development including: designing the syntax and type system, writing a parser, writing a code generator or interpreter. Furthermore, during its development a DSL also tends to grow, as new modules and data structures are found to be necessary, leading to redesigns, rewritings at the parser and the code generator. Secondly, besides the costs and difficulties of the development of a DSL, users also must be taught how to work with the DSL, for example, they must be trained to understand the language and know the different concepts. This fact may also lead to costs.

As pointed out in [48] to successfully develop a DSL, the following steps should be carefully taken into account:

- Define the problem domain and gain knowledge about the semantic notions and commonly used operations.
- Design the DSL that describes applications in this problem domain.
- Construct a support library that supports the operations and semantic notions.
- Implement the domain specific language (i.e., using an interpreter or compiler).
- Provide the environment tools needed to develop programs in the DSL (i.e., debuggers and editors).
- Provide examples, manuals and rewrite programs using the DSL.
3.3 Non-functional requirements in model-driven development

3.3.1 Functional requirement (FR) and Non-functional requirement (NFR)

A more in-depth discussion of software requirements is presented in the book [213]. In general, software requirements are partitioned into functional requirements (FR) and non-functional requirements (NFR). The FRs are associated with specific functions, tasks, features or behaviors that must be supported by the system, whereas the NFRs are constraints on various attributes of these functions or tasks. The NFRs tend to be described in terms of constraints and qualities on the results of tasks that are given as functional requirements, for example, constraints on the speed or efficiency of a given task.

Qualities are properties or characteristics of the system like maintainability, reliability, and usability which affect the degree of satisfaction with the system. Constraints are characteristics of the whole environment where the application or product will integrate and operate. For example, one constraint could be the skill set of users or developers, while another could be the characteristics of the target operating system or hardware platform. Such constraints also include performance requirements, security requirements, operating requirements and availability requirements. The software developer needs to develop strategies to address these constraints along with balancing function and service within the available resources and capabilities. Some examples of quality and constraints requirements are described as the following:

- **Availability**: the percentage of time when system is fully operational. It is an important requirement especially for time critical tasks.
- **Reliability**: the probability of the system executing without failure for a specific period of time. Normally, the quantitative reliability requirements are defined based on how serious the impact would be if a failure occurred, and whether the cost of maximizing reliability would negatively affect performance.
- **Usability**: the “ease of use” and “user-friendliness” of the product or application. For example, a developer may have to take into account that the users desire to access the application over the web using both PC and smart phone.
- **Efficiency**: how well the system utilizes the resources such as, processor capacity, disk space and memory.
- **Flexibility**: how much effort is needed to add new capabilities to a system. Flexibility is important when the system is being developed in an incremental, iterative fashion through a series of successive releases.
- **Security**: the preventing unauthorized access to a system, and protecting the privacy of data entered into (or generated by) the system.
- **Interoperability**: how easily the system can exchange data or services with other systems.
- **Robustness**: the extent a system continues to function correctly when confronted with invalid input data, defects in connected software, defects in hardware components, or unexpected operating conditions.
- **Maintainability**: the ease of correcting a defect or change in a system.
- **Portability**: the effort required to migrate a piece of software from one operating environment to another.
- **Reusability**: the extent to which a software component can be used in different applications with a minimal change.
- **Testability**: the ease with which the software components or integrated products can be tested to find defects.

Table 3.1 demonstrates the relationships between various NFR attributes. A plus sign in a cell indicates that increasing the attribute in the corresponding row has a positive effect on the attribute in the column. For example, design approaches that increase a software component’s reusability can also make the software more flexible, easier to connect to other software components, easier to maintain, more portable, and easier to test. A minus sign in a cell means that increasing the attribute in that row affects the attribute in the corresponding column negatively. For instance, efficiency has a negative impact on many other attributes. Typically, the tightest and fastest code written using a specific compiler and operating system is hard to maintain and enhance, as well as is difficult to port to another environment.
3.3 Non-functional requirements in model-driven development

3.3.2 Dealing with NFR in model-driven development

In the literature [10, 39] the authors reviewed current approaches to deal with NFRs in the context of software engineering and model-driven development in particular. For MDD, the current approaches concern: i) modeling NFRs using UML extensions or OMG standard UML profiles; ii) analyze the satisfaction of a given NFR in a particular software design by transforming this design into a specific formalism in which the analysis can take place. In [10] the authors also presented two different approaches which could be applied into a MDD framework in order to generate a software system that satisfies the desired NFRs. These approaches are illustrated in Figure 3.2 below.

In the first approach, the software developer directly modifies by hand the result of the MDD process (see Figure 3.2 a). In some cases, the developer may directly modify the code obtained after the final transformation. In other cases, the developer may be able to work at the PSM level to modify the model to satisfy the desired NFRs, and then use the model-to-model transformation (possibly modified somehow) to generate the code. Typically, this manual adaptation approach has several drawbacks:

- It takes longer to produce the software.
- It provokes lower reliability of the final product due to the human-based post-process.
- In case of changes due to maintenance, the post-process has to be replicated.

In the second approach, the developer modifies the transformation from PIM to PSM in order to obtain a PSM that satisfies the NFRs (see Figure 3.2 b). The disadvantages with the first approach are overcome, but others appear in their place.
• The complexity of the MDD framework is greater, because there are more transformations to maintain.
• The selection of the appropriate transformation (for the given set of NFRs) to apply depends on the developer, leading to a human-based pre-process.
• When the developer realizes that the available transformations are not adequate for the current process, it is necessary to build a new one making the initial configuration time longer.
4 Related work

The work presented in this thesis relates to several different research areas, where valuable work has already been performed. It is impractical to appraise related work in all these areas as part of this thesis. Instead, we focus on application development frameworks, simulation tools, and optimization approach employed in WSNs to meet application requirements.

4.1 Development frameworks

4.1.1 Middleware

The most common approach to bridge the gap between the applications and the low-level system is to develop a middleware layer mapping the higher layer into the lower one. The challenges in middleware for WSNs have been identified in [36, 64, 150, 174, 195]. Each particular middleware solution has been specialized in one or several aspects including: reprogrammability, QoS, power saving, adaptability, or aggregation functions.

A short classification of middleware frameworks is proposed in [81], where they are studied considering their adaptability to the dynamic conditions of the network and other parameters (i.e., code or data management, QoS). In this literature the authors argued that there are no approaches providing all the management tools required by sensor network applications, and as an alternative to the existing solutions they present Middleware Linking Applications and Networks (MiLAN) [81]. Another survey of middleware systems is given in [78], which presents the following classification:

- Classic middleware: provides abstractions for communication and application requirements, but in general do not consider security or QoS. Representative examples of this class including: Impala [118] and Kairos [75]. Impala supports modular programming and application adaptation in a mobile sensor network, and Kairos facilitates macro-programming model to deal with node mobility, localization and routing.

- Virtual machine (VM): is implemented on top of node operating system to provide the portability and flexibility. The VM works as a bytecode interpreter, and supports for dynamic loading of VM bytecode. This ability provides programmers with a great deal of flexibility as it allows for updating and extending a running application. Examples of WSN virtual machine are Maté [112], a TinyOS communication-centric virtual machine, and Magnet [18] a Java virtual machine.
• Data-centric middleware: emphasizes in aspects related to in-network data processing or data management (i.e., Data Distribution Service [149]), and provides database-like abstractions (i.e., Cougar [220], TinyDB [121]).

• Application-driven middleware: to address differentiated application-specific requirements at runtime such as QoS, services management or security [31, 153, 156]. MILAN [81] is an example of this class. The middleware uses service-discovery protocols to discover new nodes and notice when nodes become inaccessible. It also provides a set of network management services which is well suited to application adaptation. Another example is SensorWare [23] which uses mobile agents to collect local sensor data.

We evaluate the middleware approaches mentioned above on the basis of the following criteria: heterogeneity, scalability, power awareness, ease of use, and openness. Table 4.1 presents the evaluation results.

<table>
<thead>
<tr>
<th>Middleware approach</th>
<th>Heterogeneity</th>
<th>Scalability</th>
<th>Power awareness</th>
<th>Ease of use</th>
<th>Openness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic middleware</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impala [118]</td>
<td>Little or none</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>Kairos [75]</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
</tr>
<tr>
<td>Virtual machine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maté [112]</td>
<td>Partial</td>
<td>Full</td>
<td>Full</td>
<td>Little or none</td>
<td>Full</td>
</tr>
<tr>
<td>Magnet [18]</td>
<td>Partial</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>Data-centric middleware</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cougar [220]</td>
<td>Little or none</td>
<td>Little or none</td>
<td>Partial</td>
<td>Full</td>
<td>Little or none</td>
</tr>
<tr>
<td>TinyDB [121]</td>
<td>Partial</td>
<td>Partial</td>
<td>Full</td>
<td>Full</td>
<td>Partial</td>
</tr>
<tr>
<td>Application-driven middleware</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MILAN [81]</td>
<td>Little or none</td>
<td>Full</td>
<td>Partial</td>
<td>Full</td>
<td>Full</td>
</tr>
<tr>
<td>SensorWare [23]</td>
<td>Little or none</td>
<td>Partial</td>
<td>Full</td>
<td>Full</td>
<td>Partial</td>
</tr>
</tbody>
</table>

Table 4.1 Middleware approaches in WSN

4.1.2 Development tools

Since the applications programming for WSNs is an error-prone task, several tools have been developed in order to ease the task of building sensor network applications. In this section, the most relevant tools are described.
4.1.2.1 Visual interface between Ptolemy and TinyOS (Viptos)

Viptos [32] is an integrated graphical development and simulation environment for TinyOS applications. This framework provides an interface between TinyOS and Ptolemy II [17], which is a graphical software system for modeling, simulation, and design of concurrent, real-time, embedded systems. Ptolemy II focuses on assembly of concurrent components with well-defined models of computation that govern the interaction between components.

Viptos provides graphical development and interrupt-level simulation of actual TinyOS programs, with packet-level simulation of the network. It allows developers to create block and arrow diagrams to construct TinyOS programs from any standard library of nesC and TinyOS components. The tool automatically transforms the diagram into nesC files that can be compiled and deployed onto any target hardware. To model an application using Viptos, the developers are bound to code it for TinyOS, which implies that the developers should have sufficient knowledge of TinyOS.

4.1.2.2 Graphical Development Environment for TinyOS (GRATIS)

GRATIS [201] is built on top of Generic Modeling Environment (GME), which is a meta-programmable toolkit for creating domain-specific modeling environments. GRATIS defines a metamodel to represent TinyOS component based on the GME concepts. The three basic building blocks of GRATIS models are interfaces, modules and configurations. An interface consists of a set of events and commands. Both events and commands are functions whose return type is captured by a textual attribute. A module contains a set of interface references and its nesC code as a textual attribute. The configurations contain references to interfaces, modules and other configurations.

By implementing the metamodel above, GRATIS can parse existing nesC files and directories to build the corresponding graphical representation. From a valid model, the GRATIS code generator can transform all the interface and wiring information into a set of nesC target files. Although both wirings and code can be generated with GRATIS, but this tool was developed mainly for static analysis of TinyOS component graphs and does not support simulation.

4.1.2.3 Eclipse plugins

Eclipse (www.eclipse.org) is a popular software development environment which allows developers to create and integrate particular plugin for customizing the development environment. An Eclipse-plugin approach can benefit from existing features available within the Eclipse framework such as the powerful editor support, the persistency
4.1 Development frameworks

system and a convenient update mechanism. Several Eclipse plugins have been proposed for WSN applications programming. All of them have similar features, such as nesC syntax highlighting, code navigation, support for multiple target platforms and multiple TinyOS source trees.

TinyDT [189] is a plugin developed at Vanderbilt University. The plugin provides a custom perspective within Eclipse and has a built-in TinyOS parser which generates an in-memory representation of the actual nesC application including component hierarchy, wirings, interfaces and the JavaDoc style nesC documentation. YETI [27] is another Eclipse plugin, which consists of an optimized parser for fast execution of the system. YETI provides all important features known from development environments for other programming languages and is designed to be of use for both, un-experienced and professional sensor network developers. However, YETI still limits itself to the aspect of node programming. Other aspects regarding WSN like configuration or management are not integrated.

4.1.2.4 VeriSensor: a framework for modeling and analyzing wireless sensor networks

VeriSensor [123] consists of a domain specific modeling language and support tools to enable the modeling of a WSN by providing high level concepts that support the main use cases of WSNs. Using the framework, developers can define sensor node characteristics, how the nodes are deployed and the physical environment in which the system evolves. The framework support verifying behaviors of the node using an instantiable transition system and time Perinets.

In [123] the authors of VeriSensor present a case study where the framework is used to describe and verify a home medical monitoring application with a body sensor network. The experiment result shows that the framework is able to describe and verify sensor node behaviors regarding to the lifetime of the network. Because VeriSensor language has a formal semantic it is executable and can be simulated. However VeriSensor framework does not provide code generation solutions for any sensor node platform.

4.1.2.5 CONESC: a framework for programming adaptive WSN applications

In [2] the authors introduce a framework for programming adaptive WSN applications. The framework enables design concepts of context-oriented programming and implements the concepts in a concrete language named CONESC – an extension of nesC with the context-oriented programming constructs. The authors argue that CONESC encourages the separation of concerns in implementing adaptive WSN software because of two reasons:
• The different situations where the software needs to operate are mapped to different contexts
• The different context-dependent behaviors are encapsulated in layered functions that can be dynamically activated or deactivated

To design an application in CONESC, developers must identify the application contexts. Each context represents an environmental situation the system may encounter, and a set of behavioral variations associated to a given situation. The contexts that share common characteristic can be grouped. For example, a Battery context group may include two individual contexts: Low or Normal to describe behavioral variations of sensor node corresponding to different battery levels. An application in CONESC is simply a set of several context groups. The framework also provides developers a translator to automatically convert CONESC code into plain nesC application.

4.1.2.6 Other development frameworks

In [199] the authors described a visual development framework called WISDOM for multiplatform wireless sensor networks, which is capable of generating application code for TinyOS and Yet Another Tiny Operating System (YATOS) [200]. The visual development framework is limited to TinyOS and Yatos, since these two target platforms share the same component based programming style.

The SNACK [74] is a construction kit for sensor network applications designed to achieve reusability of applications. SNACK consists of a service configuration language, a service library, and a compiler. The language supports component declaration, component parameters, sharing state and control flows which are necessary for service construction. The service library implements the high-level networked tasks like routing trees and periodic sensing which are combined automatically into the application.

In [140] the author presents a framework for modeling, simulating and automatic code generation of WSN applications based on MathWorks tools. In the framework, WSN applications can be modeled using Stateflow state charts or Simulink block diagrams. Then the tool chains enable application developers to configure the connectivity of the sensor network nodes, perform behavioral simulations and functional verifications of the application. After modeling and simulation, this framework can generate the complete application code for several target operating systems (i.e., TinyOS or Mantis) from the simulated model.
4.2 Simulation tools

4.2.1 TOSSIM

TOSSIM [114] is a discrete event simulator integrated into TinyOS 1.x. It offers cycle-accurate low-level emulation of motes that allows the user to test and analyze the application, correct eventual errors, and run it in a controlled and repeatable environment. The core of the TOSSIM execution model is the event queue. TOSSIM events (which are different from TinyOS events) are generated by the system by commands and TinyOS events. A node identifier is assigned to each event, which allows associating the corresponding action to a node, the instant of time in which the event will be run, and a function that represents the event handler. The events are scheduled in a temporal order: from the earlier to the later. The event handler is typically an interruption handler (in the component hardware abstraction), which signals events and calls TinyOS commands.

However, TOSSIM is not the best solution in certain situations where it is necessary to measure some real world aspects. Firstly, it is not able to emulate the hardware architecture of motes. For example, a variable of type int declared in a TinyOS program, will be physically represented as an integer number in the PC architecture, typically a 32-bit field, which differs from the mote architecture representation. Secondly, TOSSIM provides a very simplistic network modeling. Although TinyViz, a GUI that permits users to interact with a network simulation, has been implemented to solve this problem, but these interactions are often ad-hoc, and TinyViz cannot reproduce complex scenarios, such as motion. Thirdly, it does not give information about timing and energy consumption because TOSSIM does not model the execution time of the CPU.

4.2.2 Avrora

Avrora [196] is a simulator of programs written for the AVR microcontroller. Currently it is able to emulate the Mica2 and MicaZ sensor nodes. Avrora’s flexible and OS independent architecture allows the monitoring of programs during their execution. It implements a high-resolution, high-fidelity simulation allowing measurement of mote performance, such as energy consumption and time execution. Avrora also emulates the behavior of the microcontroller, as well as the behavior of external devices wired to the microcontroller like flash memory. However, this simulator has some drawbacks. Firstly, it does not have GUI. Secondly, Avrora cannot simulate network management algorithms because it does not provide network communication tools.
4.2.3 Cooja

Cooja [146] is a Java simulation tool which allows a WSN application to be loaded into a simulation environment for analysis. Unlike other simulators for WSNs, Cooja is a cross-level simulator that allows for simultaneous simulations at three different levels including network level, operating system level and instruction level. At the lowest level, the instruction level, the complete node hardware platform is emulated using a discrete event microcontroller simulator. At the operating system level, nodes are simulated by executing the native operating system code with its own data memory and operating system core library. At the network level, Cooja supports simulating a network of sensor nodes with different radio models and sleep duty cycles. Extension plugins for new radio mediums and interfaces can be developed and integrated into Cooja simulation environment.

Although Cooja is developed to simulate nodes running Contiki applications, nodes running TinyOS applications can also be simulated. In this case, Cooja will use compiled binary code of TinyOS programs, and simulate the node at the instruction level. In addition, other simulators/emulators are integrated as Cooja plugins to support simulating different mote’s hardware platforms. Currently, Avrora and MSPSim [59] are integrated into Cooja. Hence, motes using AVR microcontroller and MSP430 microprocessor can be simulated now.

Cooja has a nice GUI that allows users to visualize and control the running simulation, inspecting debug message as well as radio and UART packet. A new component called the Cooja Timeline is developed to visualize both the power consumption and the network traffic during the simulation.

4.3 Optimization approaches

The efficient and robust realizations of WSNs are challenging tasks, because of the unique characteristics and several limitations of these devices. The limited availability of energy resources is the most challenging issue in a WSN, because when a mote runs out of energy it cannot participate in the network. When a WSN does not have sufficient active motes for the network application to function correctly, the entire WSN is effectively dead [50]. Therefore, WSN designs must be sufficiently optimized to ensure correct operation for at least the specified lifetime. However, as pointed out in [167], energy optimization for WSN is complex as it involves not only reducing the energy consumption of a single sensor node but also requiring dynamic tradeoffs between energy consumption, system performance, and operational fidelity.
4.3 Optimization approaches

The surveys of recent work on energy optimization in WSNs are presented in the literatures [84, 85, 108, 130, 132, 141, 143, 192, 193]. In this session we review the most relevant optimization approach at different levels (i.e., mote hardware, physical and MAC layers, network layer, OS and node software) which assist developers in meeting application-specific requirements in WSN. In general, the optimization techniques can be categorized as static or dynamic. The static optimizations optimize a WSN at deployment time and remain fixed for the whole WSN’s lifetime, whereas the dynamic optimizations provide more flexibility by continuously optimizing the WSN during runtime, providing better adaptation to the operating environment and system requirements at the same time.

4.3.1 Mote hardware platform optimization

In [56] the authors presented a modular hardware architecture for WSN mote which provides optimization opportunities at the mote hardware level via adjustable parameters (i.e., processor voltage and frequency, sensing frequency, duty cycle, etc.). In a modular design, the adjustable parameters, whose values can be specialized to meet varying application requirements, associate with various main components of a WSN mote such as power unit, storage unit, sensing unit, processing unit and transceiver unit. For example, the sensing unit’s adjustable parameters can control power consumption by changing the sensing frequency (i.e., constant sensing, periodic sensing, or sporadic sensing) and the resolution of the analog to digital converter. The processing unit’s adjustable parameters include processor voltage and frequency, which can be specialized to meet power budget and throughput requirements. The transceiver unit’s adjustable parameters include modulation scheme, data rate, transmit power, and duty cycle, whose values can be customized for the purpose of energy saving.

As pointed out in [38], wireless communication remains the most energy-consuming operation. Sending a single bit of information 100 meters may consume more energy than 1000 CPU instructions. The communication cost is incurred by both the sender and receiver node, and by any intermediate nodes in multi-hop paths which may grow as the network becomes larger. Therefore, energy savings must be driven by energy-aware design throughout the network communication stack, rather than relying on improvements in hardware technology.

4.3.2 Physical layer and MAC layer optimization

The literature [84] focuses on the setting of the physical layer parameters to solve the problem of energy-efficient transmission of data over a noisy channel. In this work the authors investigated the relationship between transmit energy and other physical layer
parameters such as hop distance and modulation scheme. An interesting result is that because the transmit energy must be proportional to $d^n$ where $n \geq 2$ and $d$ is the distance between the transmitter and receiver, the network routes containing many short-distance hops may be more energy-efficient than routes containing few long-distance hops. In addition, the authors proposed a metric called the energy per successfully received bit, which specifies the expected energy required to transmit a bit successfully over a particular distance given a channel noise model. This metric is a function of three physical layer parameters: hop distance, transmit energy, and the modulation scheme. By minimizing this metric, the developer can select the hop distance, transmit power and/or modulation scheme that maximize network lifetime.

At the MAC layer, WSN developers can tune different parameters such as channel access schedule, message size, and duty cycle to meet application requirements [160]. In some MAC protocols, the developers can adjust wireless channel slot allocation to optimize throughput while maintaining the traffic load balance between sensor nodes. A fairness index measures the load balancing ratio of packets delivered to the sink node from all the senders. For the ideal load balance, the fairness index is 1. Example of these protocols include Traffic Adaptive Medium Access Protocol (TRAMA) [169], Berkeley Media Access Control (B-MAC) [161], and Zebra MAC (Z-MAC) [93]. In other approaches, MAC protocols can adapt their duty cycle, message size, and transceiver operating modes to reduce power consumption. For instance, Sensor MAC (S-MAC) [208] tunes both the duty cycle and message size for energy saving.

4.3.3 Network layer optimization

The main task of sensor nodes in a WSN is to sense and collect data about a phenomenon and transmit the data to the sink node. To meet application requirements, this data dissemination requires energy-efficient routing protocols to establish communication paths between the sensor nodes and the sink. Ideally, data dissemination and routing protocols should target energy efficiency, robustness, and scalability. To achieve these optimization objectives, routing protocols adjust transmission power, routing strategies, network topologies and employ either single-hop or multi-hop routing.

For example, Sensor Protocols for Information via Negotiation (SPIN) [107] allows sensor nodes to adjust routing activities according to available energy resources. There are two routing phases in SPIN, data negotiation and resource adaptation. In the data negotiation phase, nodes exchange metadata before the starting the actual data transmission. The metadata contains brief information about node’s actual data. Based on the metadata, the nodes which are interested in the data content will request the
actual data. The data negotiation ensures that data is sent only to the interested nodes. At the resource adaptation phase, depend on the energy levels, sensor nodes may reduce or eliminate certain activities (i.e., forwarding of metadata and data packets).

To meet the strict real-time requirements for sensed data delivery, the routing protocol SPEED [194] tries to optimize data report velocity of sensor nodes in a WSN. In SPEED, the data report velocity value and the next-hop relay node is selected based on a pre-defined single-hop delay guarantee value.

In the LEACH protocol [82], node’s radio transmission power can be customized to control network topology (based on routing paths) for energy saving. LEACH uses a hybrid single-hop and multi-hop communication paradigm. The sensor nodes use multi-hop communication to transmit data reports to a cluster head (LEACH determines the cluster head using a randomized distributed algorithm). The cluster head forwards the aggregated data to the sink node using long-range radio transmission.

4.3.4 Application layer dynamic optimization

Dynamic optimization approaches enable sensor nodes to dynamically customize their parameter values in situ according to both operating environment and application requirements. In the literature [57] the authors mentioned several generic techniques that can be implemented for the application layer or for specific subsystems to provide dynamic optimization for the purpose of energy saving. The most common techniques are:

- **Duty-cycling**: Application can control the duty cycle so that sensor nodes are powered in a cyclic manner to reduce the average power draw.
- **Batching**: Multiple operations are buffered and then executed in a burst to reduce startup overhead cost.
- **Hierarchy**: Application can arrange operations in a hierarchy based on energy consumption and then invoke low energy operations before high energy operations.
- **Redundancy reduction**: Application can reduce redundancy by compression, data aggregation, and/or message suppression.

In [104], the authors proposed an approach for software reconfiguration in WSNs based on model-integrated computing. The authors modeled the WSN operation space (defined by the WSN software components’ models and application requirements) and defined reconfiguration as the process of switching from one point in the operation space to another. In that way, the key question in reconfiguration of WSNs, which is how to decide what the new configuration should be, is considered as a search problem in...
the operation space. Another dynamic optimization approach for WSNs is presented in [142]. The authors implemented Markov Decision Process to dynamically optimize mote’s processor voltage, frequency, and sensing frequency in accordance with application requirements over the lifetime of the motes. The main drawback of the dynamic optimization approaches is the optimization module should be implemented on each sensor node with constrained resources.
Part II
Solution
5 Programming model

In this chapter, we describe a rule-based programming model which is adopted in our proposed solution. In the first subsection, we present the current state of the art in programming approaches for WSNs. In the second subsection, two kinds of models are analyzed: node-centric and macro-programming. The first one considers the programming of an individual sensor node, while the second approach is focused on describing the behavior of a WSN as a whole. The rule-based programming model is described in the last subsection of the chapter.

5.1 Programming approaches for WSNs

In the last few years, much work has targeted the development of programming support in the effort to meet WSN application requirements. Figure 5.1 illustrates the taxonomy which is used to classify the different programming approaches appeared for WSNs. An exhaustive classification is presented in the literatures [139, 187].

![Figure 5.1 Classification of programming approaches for WSNs](image)

5.1.1 Programming languages

A programming language is used to develop an application for WSNs. There are two major categories:

- **Imperative languages**: This class of programming approaches employs purely imperative languages with sequential or event-driven semantics. Currently, C and nesC [113] are the most popular languages for WSNs.
• **Declarative languages:** Declarative language is known to encourage programmers to focus on program outcomes (what a program should achieve) rather than implementation (how the program works). In the context of WSN, this language may provide a general-purpose, easy to use and efficient programming approach to developers. Existing solutions in this category includes database-style, or rule-oriented programming model, such as TinyDB [121], Cougar [220] or FACTS [191].

### 5.1.2 Programming models

Programming models in WSN represent the different proposals to deal with the special features and complexity of the applications programming. Programming models provide abstractions at different architectural levels to ease the programming task. For example a programming model can liberate the programmer from having to address the low-level WSN mechanisms like messaging and routing protocols, data caches and neighbor lists. In WSN, there are two main programming models namely node-centric and macro-programming.

The node-centric (or local behavior) model focuses on the programming of individual sensor nodes. Hardware abstraction layers and operating systems have been provided to support programming. As opposed to traditional systems, the underlying operating system decides the programming language to be used. Operating systems also present different programming languages (imperative, declarative, objects) which increase the heterogeneity and make the portability difficult.

The macro-programming (or global behavior) approach allows to program the network as a whole. This model provides abstractions and supports to express the distributed processing of sensor nodes in the network. Macro-programming model often requires a middleware, which is implemented in each individual node, to hide the low-level node-centric programming and to support node co-operations.

### 5.2 Node-centric and macro-programming approaches

#### 5.2.1 Node-centric model

In node-centric model, the developer has to translate the global application behavior into the local actions of each node in the network, and individually program the node. The application can be written on the top of WSN operating systems (OS-based), which are designed to provide the hardware abstractions. Although it is possible to obtain the hardware independence in some degree, the operating system independence has not been completely achieved:
The programming language for WSN applications building is limited to the used for writing the underlying operating system. Subsequently, applications using TinyOS are forced to be developed in nesC and applications using Contiki, Mantis or LiteOS are developed in C/C++ programming language.

The execution models and programming paradigm are exported from the underlying operating system to the application level.

In terms of communication and collaboration between nodes, the node-centric model gives programmers basic communication facilities in which a node can exchange messages only with the surrounding nodes located within its communication radius (physical neighborhood). The communication protocol can be broadcast (messages are sent to all reachable nodes) or unicast (messages are sent to a specific reachable node). In general, the node identifier is used to assign the address of the node in the network.

In some cases, middleware or a virtual machine may be used to support the development, maintenance, deployment and execution of applications, filling in the gap between the application layer and the hardware, operating system and network stack layers.

5.2.2 Macro-programming model

Macro-programming involves programming the network as a whole, rather than writing software to drive individual nodes. This approach introduces a new and completely different view on how to program WSNs. The developer defines the global behavior of the WSN at a high-level specification, and the local behaviors are automatically generated on each node. This relieves application developers from having to deal with concerns at each network node.

Macro-programming can be realized at the group-level or network-level. In the group-level, the developer can handle a subset of nodes, as well as a set of operation on this subset. The group is defined based on physical closeness or logical properties. For example, a neighborhood-based group consists of a node and its neighbors, whereas a logical group consists of nodes that are sharing the same properties such as node type or sensor reading values. At the network-level, a sensor network is treated as a whole, and is regarded as a single abstract entity (i.e., a database).

5.3 Rule-based programming model

It is really difficult to the research community to conclude which is the best approach to programming sensor networks to meet various application requirements. In this thesis we employ a rule-based approach which consists of a declarative language and a node-
5.3 Rule-based programming model

centric programming model. A rule-based approach is chosen because of the advantages that this approach has over alternatives. In [179, 210] the authors explained that the programming and concurrency models of a rule-based approach are simplified compared with other approaches. They believed that the simpler execution model will make it easier to prove programs correct and minimize power usage while still being sufficiently expressive to capture diverse high level abstractions. Another benefit, according to [191], is that thinking about the system from a declarative paradigm applies better to sensor networks than thinking about the system using an imperative paradigm. Therefore, the authors argued that rule orientation is a more natural way to express programs for sensor networks. In the next subsections we describe the programming model in details.

5.3.1 Node behaviors

Our programming model is based on the node-centric model described above. In order to describe the local behaviors of the sensor node we employ a set of flexible and extensible rules. These rules are used to evaluate incoming messages of each sensor node and ask this node to perform the corresponding actions. Figure 5.2 depicts the working behaviors of a single node in the WSN.

In our approach, a node may receive messages from its attached sensors or from the neighbor nodes located within its communication radius. Two communication protocols are supported including broadcast and unicast. For unicast protocol, node identifier is used to assign the address of the node in the network. Each node maintains a message buffer, and the incoming messages are automatically stored in this buffer. A set of pre-
defined rules is used to process the messages in the buffer. Each rule consists of three parts:

- **SELECT**: The *SELECT* clause includes a logical *CONDITION* to select all messages which satisfy this condition from the message buffer.
- **PROCESSING**: The *PROCESSING* clause is used to define the necessary in-network processing activities, i.e. data aggregation. These activities will be applied to the set of selected messages created by the *SELECT* clause. Since many applications do not perform in-network processing, *PROCESSING* is an optional part of the rule.
- **ACTION**: The *ACTION* clause defines the set of actions that will be performed to the final set of message such as forwarding, modifying or dropping.

Each rule is executed periodically (after a time $\Delta T$) by a rule interpreter. The rules that need to be executed simultaneously are grouped into a rule set. An application may contain several rule sets with different values of $\Delta T$.

### 5.3.2 Message structure

Our programming model focuses much on data dissemination and processing, which is suitable for a WSN using data-centric communication protocols. Such a protocol is presented in [95, 105], in which exchanged messages carry all necessary information to allow data handling and processing without further knowledge (i.e., about the sender’s address or network topology). In our approach, the message is encoded using attribute-value pairs. The common message structure is described as the following:

$$\text{Message} = \{(\text{attribute1}, \text{value1}), (\text{attribute2}, \text{value2}), \ldots, (\text{attributeN}, \text{valueN})\}$$

For each message, at least one attribute and the corresponding value are needed. Additional attribute-value parameters can be added in order to provide meta-information about the contained data. Examples for such messages are:

- **Message** = \{\text{temperature, 50}, \text{node1d, 1}\} – a temperature of 20°C is measured by the sensor node #1 of the WSN.
- **Message** = \{\text{object, 1}, \text{locationx, 50}, \text{locationy, 100}\} – an object is detected at the coordinates [50,100].

### 5.3.3 Rule structure

As mentioned above, a rule consists of three different parts: a *SELECT* clause, a *PROCESSING* clause, and an *ACTION* clause. These parts are described in the following subsections.
5.3.3.1 The SELECT clause

The SELECT clause includes a logical CONDITION to select all messages which satisfy this condition from the message buffer. All selected messages will be stored in a temporary buffer for later processing. The parameters of the CONDITION can be the attributes of received messages, the attributes of the local node, application’s parameters, or the pre-defined functions in the rule interpreter library. Table 5.1 demonstrates several examples of the SELECT clause. Table 5.2 shows the list of possible application parameters, node attributes and pre-defined functions. The list may be extended later, depending on the application domain.

<table>
<thead>
<tr>
<th>Select clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT: (1)</td>
<td>Select all messages in the buffer</td>
</tr>
<tr>
<td>SELECT: (temperature)</td>
<td>Select all messages which report a measured temperature value.</td>
</tr>
<tr>
<td>SELECT: ((object==1)&amp;&amp;(nodeid==1))</td>
<td>Select all messages from node #1 which detect an object.</td>
</tr>
<tr>
<td>SELECT: (hopcount&gt;MAXHOPCOUNT)</td>
<td>Select a message if the hopcount is greater than a threshold. MAXHOPCOUNT is an application’s parameter which is defined by the developer.</td>
</tr>
<tr>
<td>SELECT: (destinationid==$hostd)</td>
<td>Select all messages are sent to a node using unicast protocol. The message’s attribute destinationid is the address of the destination in the unicast sending. The $hostid is a local node’s attribute expressing the host address.</td>
</tr>
<tr>
<td>SELECT: ($probability()&lt;0.5)</td>
<td>Randomly select a message. $probability() is a pre-defined function in the rule interpreter library which returns a random value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.1 Examples of SELECT clause</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node attributes</strong></td>
</tr>
<tr>
<td>$hostid</td>
</tr>
<tr>
<td>$total_msg_count</td>
</tr>
<tr>
<td>$selected_msg_count</td>
</tr>
<tr>
<td>$position_x, $position_y</td>
</tr>
<tr>
<td>$time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Application’s parameters</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFERLENGTH</td>
<td>Maximum number of messages can be stored in the MessageBuffer</td>
</tr>
</tbody>
</table>

80
 QUEUELENGTH: Maximum number of messages can be stored in the Sending buffer.

 MAXHOPCOUNT: The total number of hops permitted for packets traversing the network.

 MAXLIFETIME: The maximum permitted time for a packet to remain in transit.

 GOSSIP_K: Parameter K in the GOSSIP(P,K) protocol [77]

 GOSSIP_P: Parameter P in the GOSSIP(P,K) protocol [77]

 TEMP_THRESHOLD: A temperature threshold in a fire detection application

<table>
<thead>
<tr>
<th>Pre-defined functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>probability()</td>
<td>Return a random value for probabilistic decisions</td>
</tr>
</tbody>
</table>

Table 5.2 List of possible node attribute, application parameters, and pre-defined functions

5.3.3.2 The PROCESSING clause

The clause is used to define the necessary in-network processing activities which are applied to the set of selected messages created by the SELECT clause. PROCESSING clause is an optional part of the rule. Table 5.3 presents a list of current in-network processing function supported by our model. The list may be extended later, depending on the application requirements.

<table>
<thead>
<tr>
<th>In-network processing functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Average(attribute)</td>
<td>Calculate and return the average of the attribute values. For example function @Average(temperature) will calculate and return the average temperature value of all selected message.</td>
</tr>
<tr>
<td>@Max(attribute)</td>
<td>Find and return the maximum of the attribute values.</td>
</tr>
<tr>
<td>@Min(attribute)</td>
<td>Find and return the minimum of the attribute values.</td>
</tr>
<tr>
<td>@Median(attribute)</td>
<td>Calculate and return the median of the attribute values.</td>
</tr>
<tr>
<td>@Sum(attribute)</td>
<td>Calculate and return the sum of the attribute values.</td>
</tr>
</tbody>
</table>

Table 5.3 List of supported in-network processing functions

5.3.3.3 The ACTION clause

The ACTION clause defines the set of actions that will be performed to the final set of message such as forwarding, modifying or dropping. A list of currently available action supported by our model is presented in Table 5.4. This list may be extended later, depending on the application requirements.
## 5.3 Rule-based programming model

<table>
<thead>
<tr>
<th>Rule’s actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Send(com_id, protocol)</td>
<td>Forward current message using a specific communication interface and protocol.</td>
</tr>
<tr>
<td>@Drop()</td>
<td>Discard current message.</td>
</tr>
<tr>
<td>@DropAll()</td>
<td>Discard all selected message.</td>
</tr>
<tr>
<td>@Create(attribute, value)</td>
<td>Create a new message with the pair attribute-value and put the message to the buffer.</td>
</tr>
<tr>
<td>@Modify(attribute, value)</td>
<td>Modify the pair attribute-value of a message then put it back to the message buffer.</td>
</tr>
<tr>
<td>@Return()</td>
<td>Return current message to the message buffer.</td>
</tr>
<tr>
<td>@ReturnAll()</td>
<td>Return all selected message to the message buffer.</td>
</tr>
</tbody>
</table>

Table 5.4 List of available rule actions
6 Software architecture

This chapter presents the description of the sensor node software architecture. The architecture aims to abstract away the hardware heterogeneity to support writing applications independently of the platform. It is designed in a multi-layered fashion, which allows focusing on details of each architectural component. To achieve this goal, firstly we analyzed a typical WSN mote architecture focusing on functionality, interface, and structure of the hardware components and the operating system of WSN motes. Then we modeled the hardware components and the operating systems like a black box connecting applications and hardware devices: it exports a service interface (input) and transforms it into hardware requests (output). To describe the hardware devices integrated into sensor nodes, identifying their main features and operations, the XML Manifests files are also used, conforming to the specification described through XML Schemas (or DTDs).

In addition, the architecture design is elaborated using implementation diagrams of UML 2.0. The mathematical descriptions of the architecture are established to allow describe each layer without ambiguity. The PIMs and PSMs have been created to be able to define particular instances of the applications, operating system and hardware components. An overview of the software architecture design, architecture formalization, hardware layer, operating system layer as well as the corresponding PSMs, are described in this chapter. The following chapters describe both the application layer and the operating system abstraction layer in detail.

6.1 Architecture design

The main motivation of the proposed architecture is to support addressing the portability and application development challenges for WSN programming. With this goal in mind, the following design principles are considered:

- **Using the node-centric model**: The architecture is focused on programming generic individual sensor node in a WSN (following the node-centric approach). A generic sensor node is considered as a self-contained device presenting a set of common features at different abstraction levels.
- **Multi-platform**: The architecture is hardware and OS independent supporting a reasonably broad set of platforms and OSs.
- **Multi-layer design**: Every sensor node abstraction level is modeled as a component or layer, which encapsulates certain functionalities and carries clear responsibilities. A layer can interact with the immediate upper and lower one through a predefined interface, which is provided and used respectively. It
means that every layer is itself a connector between both interfaces. Additionally, to ensure an independent architecture, layers are interchangeable among different instances of specific sensor nodes.

- **Programming abstractions**: Applications are built on top of the architecture independent of the underlying platform, and a translator/compiler over the OS layer deals with the complexity of the translations between both abstraction levels.

- **Generation of code**: This process takes a generic application (native application) as input and transparently transforms it into the equivalent platform-specific application (final application). In this way, the tailored code is automatically generated, which can be also compiled for a particular sensor node to obtain the executable code. The same single generic application could be translated to different platforms.

- **Reusability**: The proposed architecture should be able to reuse the basis of the original architecture when possible. In particular, existing hardware and operating systems should be taken into account. Moreover, every component in the sensor node architecture could be reused, and the integration should be supported.

![Figure 6.1](image_url) The proposed software architecture for WSN mote

Figure 6.1 presents the proposed software architecture for WSN motes programming. Taking into account all above requirements, the architecture is designed using a MDA-based architecture. In this way, a platform-specific sensor node can be viewed as a
particular instance of a generic sensor node. To obtain high-level abstractions, a DSL is specified to support developing applications according to the proposed rule-based model. At the lower levels, an Operating System Abstraction Layer is incorporated, in order to carry out transformations of DSL applications into OS-specific applications. The translations are not limited to the interfaces conversions, but a complete adaptation is required to support different WSN operating systems and hardware platforms. The output will be tailored code that will be deployed to the target platform. The advantages of this approach can be summarized as follows:

- Ease of applications programming by a high-level interface friendly to applications.
- Application portability, due to the application developed being transparently translated to the underlying platform with no additional work.
- Clear and platform-independent applications design, incorporating the required knowledge while specific details are kept hidden.
- Flexibility to incorporate new components and ease of integration. Components are interchangeable with other analogous ones in the architecture.

![Component diagram of the mote architecture](image)

The architecture above can also be represented using an UML 2.0 implementation diagram in Figure 6.2, in which every layer is encapsulated into a component. Every component provides a predefined interface to the upper layer and uses the interface offered by the immediate lower level. As represented in Figure 6.2, a generic
architecture is divided at different layers: every component can be instantiated to compose a specific architecture. For example, instances of the OS layer could be TinyOS, Contiki, or any WSN operating system which provides the proper interface. Similarly, Zolertia Z1, Mica or TmoteSky sensor nodes could be the instances of the hardware layer. In the subsequent sessions, the implementation of each layer/component is described in detail.

### 6.1.1 Implementation of the Hardware layer

Figure 6.3 illustrates an implementation diagram representing the Hardware Layer (HL). It includes the physical devices held in sensor nodes, and the main settings identified. Every component has a specific interface, related to the functionality associated with the physical device. In this layer, the interfaces provide the lowest-level services in the architecture. Conceptually, the whole set of services provided by the hardware layer interface \( HLI \) is composed of the union of the sub-interfaces provided by the physical components, which can be expressed in Equation 6.1 below:

\[
HLI = HLI_{microcontroller} \cup HLI_{sensorboard} \cup HLI_{radio} \cup HLI_{serial} \cup HLI_{bus} \cup HLI_{flashmemory} \cup HLI_{leds}
\]  
\( \text{Equation 6.1} \)

We employed XML to describe the hardware components of a sensor node. To support validating the XML files, DTD schema is employed [203]. One corresponding DTD file is created for each type of component (i.e., CPU, radio, sensor board or sensor) in order to completely characterize its features, operations and restrictions. Every instance of a hardware component type is described through a XML file which contains the characteristics, interfaces and compatibilities of the particular hardware. The XML file must be manually created by the programmer in accordance to its DTD. The detail of the components in the Hardware layer is described in the session 6.3.

### 6.1.2 Implementation of the Operating System layer

Figure 6.4 presents the design of the Operating System layer (OSL), which is composed of several components encapsulating the functionalities derived from the physical hardware devices.
Figure 6.3 Implementation diagram of the Hardware Layer
The set of services provided by this layer is the union of all the functionality provided by the integrated components. The set of interfaces (OSLI) could be expressed as following:

\[
OSLI = OSLI_{\text{scheduling}} \cup OSLI_{\text{clock}} \cup OSLI_{\text{sensor and actuator}} \cup OSLI_{\text{communication}} \cup OSLI_{\text{storage}} \cup OSLI_{\text{leds}}
\]  

Equation 6.2

6.1.3 Implementation of the Operating System Abstraction layer

The Operating System Abstraction Layer (OAL) is an intermediate layer which supports abstracting away the WSN operating system. The layer’s objective is to translate the written application using a DSL into the WSN OS-specific implementations (i.e., program code in TinyOS or Contiki), which can be then translated into binary code by the target compiler. Figure 6.5 depicts the implementation diagram of the OAL. As shown, the layer is composed of four main components: OAL Interface, Rule Interpreter, OS Translator Interface, and OS Translator. The first component provides the translation services to the upper layer. The second one is used to convert the PIM application, which is created using the DSL and rule-based programming model, into an intermediate program code. This program code is constructed using a set of OS independent functions.
listed the OS Translator Interface. The last one, OS Translator, is intended to parse the intermediate code and generate the equivalent code for a target platform.

![Diagram showing OAL Services, Rule Interpreter Services, OS Translator Services, and OSL Services]

Figure 6.5 Implementation diagram of the Operating System Abstraction Layer

The Rule interpreter layer provides a one-one mapping from a PIM model to an intermediate program code. The OS Translator layer (OSTL) should offer the set of constructs and functionality which are obtained from the analysis of different WSN operating systems interfaces. Subsequently, the set of OSTL interfaces (OSTLI) should encapsulate the specific OSs services, and can be expressed as following:

\[
OSTLI = OSTLI_{\text{scheduling}} \cup OSTLI_{\text{clock}} \cup OSTLI_{\text{sensor and actuator}} \\
\cup OSTLI_{\text{communication}} \cup OSTLI_{\text{storage}} \cup OSTLI_{\text{leds}} 
\]

Equation 6.3

Note that the composition of the OSTL interfaces should be the same as that offered by the operating system in order to obtain portability. The components OS\(i\) Translator is responsible for mapping these functions to the OS\(i\)-specific implementation.
6.1.4 Implementation of the Application layer

Figure 6.6 illustrates the implementation diagram of the highest layer, the Application Layer (APL). This layer represents the complete domain of WSN applications that are able to interact with the proposed architecture. The layer consists of an Eclipse Plugin component which interacts with the Eclipse IDE platform to provide user a friendly graphical IDE. The DSL model library, simulator interface and optimization interface are also included into this layer.

![Diagram of Application Layer Implementation](image)

**Figure 6.6 The implementation diagram of the Application Layer**

6.2 Architecture formalization

We utilize set theory to formally describe the proposed architecture. The architecture formalization provides an explanation of how to assemble the concrete software architecture of a particular WSN mote from a set of components and layers defined in the proposed architecture above, and to represent the set of potential instances of architectures. A concrete architecture is composed of four layers: hardware, OS, OAL and application, which are formalized as follows:

Let $A$ be the set of WSN software architectures which can be defined as the Cartesian product of the set of WSN hardware platforms, operating systems, the abstraction layers, and the applications:

$$A = APP \times OAL \times O \times H = \{(a, b, c, d) | a \in APP \land b \in OAL \land c \in O \land d \in H\}$$

Equation 6.4
where \textit{APP} represents the potential set of applications. \textit{OAL} is corresponding to the set of abstraction defined on the WSN operating system. \textit{O} is the set of operating system designed for motes:

\[ O = \bigcup O_i, \forall i = 1..n \quad \text{Equation 6.5} \]

\( H \) is the set of sensor motes which can be defined as the Cartesian product between platforms \( P \) and sensor devices \( S \) (refer to Section 2.1):

\[ H = P \times S = \{(a, b) | a \in P \land b \in S\} \quad \text{Equation 6.6} \]

For example:

\( APP = \{\text{Blink, Surge, Oscilloscope, ...}\}; \text{OAL} = \{\text{Rule translation}\} \)

\( O = \{\text{TinyOS, Conkiti, MantisOS, ...}\}; \text{P} = \{\text{Zolertia Z1, TmoteSky, Mica, MicaZ, ...}\}; \text{S} = \{\text{mts300, mts310, mda300, ... } \} \}

From an architectural point of view, each one of these sets (except the \textit{APP}) can be described as the union of two subsets: functionality (\( F \)) and interface (\( I \)). Thus:

\[ H_i = \{HF_i \cup HI_i\}, \forall H_i \in H \quad \text{and} \quad O_j = \{OF_j \cup OI_j\}, \forall O_j \in O \]

\( OAL_k = \{OALF_k \cup OALI_k\}, \forall OAL_k \in OAL \quad \text{Equation 6.7} \)

Note that the interface of the \( OAL_k \) is denoted as \( OALI_k \), and it corresponds to the interface provided by the component \( OS_k \) Translator \( OSTLI_k \) in the OS Translator layer (OSTL). Now, let \( R_T \) be defined as the translation relation over the Cartesian product of \( OSTL \times O \). Then, \( R_T \subseteq OSTL \times O \), such that:

\[ OSTL \times O = \{(OSTLI_i, O_j) | OSTLI_i \in OSTL \land O_j \in O\} \]

\[ R_T = \{(a, b) \in OSTL \times O | \forall x \in OSTLI_a \exists y \in OI_b,\ OSTLI_{ax} \text{ is semantically equivalent } OI_{by}\} \]

\[ OSTLI_i R_T O_j \iff (OSTLI_i, O_j) \in R_T \quad \text{Equation 6.8} \]

Given such translation relation \( R_T \) between the sets \( OSTL \) and \( O \), it is possible to define \( f_T \) as the function to compute the mapping between the OS Translator Interface \( OSTLI_i \) and the Operating System Interface \( O_j \). Then, it is denoted as \( f_T : OSTLI_i \rightarrow OI_j \) and it satisfies:

\[ f_T(x) = y, \forall x \in OSTLI_i, \exists y \in OI_j, OSTLI_i \in OSTL \land O_j \in O \mid OSTLI_i R_T O_j \quad \text{Equation 6.9} \]
Note that the interface provided by the component $OS_i$ Translator in the OS Translator layer $OSTL_i$ corresponds to the interface $OAL_i$ of the $OAL_i$.

Similarly, at the underlying level, let $R_p$ be a portability relation defined over the Cartesian product of $O \times H$. Then, $R_p \subseteq O \times H$, such that:

$$O \times H = \{(O_i, H_j) | O_i \in O \land H_j \in H\}$$

$$R_p = \{(a, b) \in O \times H | \forall x \in O \exists y \in H, O_a, x \text{ is semantically equivalent } H_b, y\}$$

$$O_i \mathcal{R}_p H_j \iff (O_i, H_j) \in R_p$$ \hspace{1cm} \text{Equation 6.10}

Let $f_p$ be the function to compute the mapping between the Operating System Interface $O_i$ and the Hardware Interface $H_j$. Then, it is denoted as $f_p: O_i \rightarrow H_j$ and it satisfies:

$$f_p(x) = y, \forall x \in O_i, \exists y \in H_j, O_i \in O \land H_j \in H | O_i \mathcal{R}_p H_j$$ \hspace{1cm} \text{Equation 6.11}

In this work we focus on the transformation between the Operating System Abstraction Layer, specially the OS Translator Layer, and the Operating System Layer, where the OS-specific code is generated from the PIM model. Based on Equation 6.9 we can define the algorithm which is used to create the mapping rules between the $OSTL_i$ and the $O_j$. Such algorithm is presented in the Listing 6.1 An algorithm for generating the mapping between Operating System Translator Layer and Operating System Layer

### An algorithm for generating the mapping between $OSTL_i$ and $O_j$ $f_T: OSTL_i \rightarrow O_i$

```plaintext
for all $x$ in $OSTL_i$ do
    $f_T(OSTL_i,x) = \emptyset$
for all $y$ in $O_i$ do
    if $x$ is semantically equivalent $y$ then
        $f_T(OSTL_i,x) = O_i,y$
endfor
```

Listing 6.1 An algorithm for generating the mapping between Operating System Translator Layer and Operating System Layer

### 6.3 Hardware layer

This section focuses on identifying the main components at the hardware level with their settings and functionality. An interface has also been determined for each device. However, it is important to point out that the OS is responsible for supporting a specific hardware. We assume that the mentioned hardware platforms in this section are currently supported by the WSN OSs.
6.3.1 Hardware component description

We elaborate Document Type Definition (DTD) and XML files in order to specify different hardware platforms. For each type of component (i.e., CPU, radio, sensor board or sensor), one corresponding DTD file is created to characterize its features, operations and restrictions. Every instance of a hardware component type is described through a XML file which contains the characteristics, interfaces and compatibilities of the particular hardware. The XML file must be manually created by the developer in accordance to its DTD. Listing B.1 - Appendix B demonstrates an example of DTD file which is used to specify a digital temperature sensor. According to this DTD file, a XML file is created to describe a particular digital temperature sensor named TMP102 on the Zolertia Z1 mote [225]. This XML file is presented inListing 6.2 A XML file to describe Digital Temperature sensor TMP102 on Z1 Mote

A XML file to describe Digital Temperature sensor TMP102 on Z1 Mote

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SENSORSYSTEM "dtd_digital_temperature_sensor.dtd">
<SENSOR>
  <Manifest Signature="TMP102-AIDRLT-0000001">
    <definition datasheet="http://www.ti.com/product/tmp102">
      <model>TMP102</model>
      <vendor>Texas Instruments</vendor>
      <type>Digital</type>
      <sensorboard>
        <board model="Z1MSP430"/>
      </sensorboard>
      <reading>
        <units name="degrees" out="digital"/>
        <scale size="0.5"/>
        <voltageout voltagein="" constant="" variable="tmp102.pdf"/>
        <voloutvalue min="0" max="0.4"/>
        <tempvalue min="-25" max="85"/>
      </reading>
    </definition>
    <signal>
      <output name="I2C"/>
      <location name="I2C"/>
      <power name="TEMP_PWR"/>
    </signal>
    <assemblies>
      <file name="SensorTest" provider="Zolertia(TM)" fileAssembly="examples/z1/test-tmp102.c"/>
    </assemblies>
    <interface>
      <export>
        <operation>
          <action name="tmp102_init" fileAction="platform/z1/dev/tmp102.c" ReturnValue="error">
            <parameter index="0" type="signature"/>
          </action>
          <action name="tmp102_read_temp_simple" fileAction="platform/z1/dev/tmp102.c"/>
        </operation>
      </export>
    </interface>
  </Manifest>
</SENSOR>
```
6.3 Hardware layer

The set of Hardware Layer Interfaces (HLI) is composed of the union of the sub-interfaces provided by the physical components such as microcontroller, sensor board, radio, I/O bus, flash memory, ... (refer to Equation 6.1). The details of various hardware components were described in Chapter 2, section 2.1. Table 6.1 expresses the hardware layer interface set which cover the components’ functionalities.

<table>
<thead>
<tr>
<th>Hardware component</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microcontroller</strong></td>
<td></td>
</tr>
<tr>
<td>RAM memory</td>
<td>readMemory(int *dir, char *buf, int length)</td>
</tr>
<tr>
<td></td>
<td>writeMemory(int *dir, char *buf, int length)</td>
</tr>
<tr>
<td>ROM memory</td>
<td>readByte(int *dir, char *byte)</td>
</tr>
<tr>
<td></td>
<td>writeByte(int *dir, char *byte)</td>
</tr>
<tr>
<td>Clock</td>
<td>startTimer(int timer, int interval)</td>
</tr>
<tr>
<td></td>
<td>stopTimer(int timer)</td>
</tr>
<tr>
<td></td>
<td>getCounter()</td>
</tr>
<tr>
<td></td>
<td>setCounter(int value)</td>
</tr>
<tr>
<td>Power management</td>
<td>adjustPower()</td>
</tr>
<tr>
<td></td>
<td>setMode(int mode)</td>
</tr>
<tr>
<td></td>
<td>int getMode()</td>
</tr>
<tr>
<td>ADC</td>
<td>initialize()</td>
</tr>
<tr>
<td></td>
<td>analogToDigital(int channel, int port)</td>
</tr>
</tbody>
</table>

**Internal Bus**

| UART                | initialize() |
|                     | deinitialize() |
|                     | sendByte(char *byte) |
|                     | receiveByte(char *byte) |
|                     | sendFrame(int cmd, int payload_length, char *payload) |
|                     | receiveFrame(int cmd, int *payload_length, char *payload) |

| I2C                  | enable() |
|                     | disable() |
|                     | int start() |
|                     | stop() |
|                     | read(char *byte) |
**Communication**

<table>
<thead>
<tr>
<th>Component</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio</td>
<td>init()</td>
</tr>
<tr>
<td></td>
<td>on()</td>
</tr>
<tr>
<td></td>
<td>off()</td>
</tr>
<tr>
<td></td>
<td>send(char *payload, int payload_length)</td>
</tr>
<tr>
<td></td>
<td>receive(char *payload, int *payload_length)</td>
</tr>
<tr>
<td></td>
<td>channel_clear()</td>
</tr>
</tbody>
</table>

| Serial    | initialize() |
|           | deinitialize() |
|           | sendByte(char *byte) |
|           | receiveByte(char *byte) |
|           | sendFrame(int cmd, int payload_length, char *payload) |
|           | receiveFrame(int cmd, int *payload_length, char *payload) |

**Sensor Board**

| Sensor | int sample(char *buffer) |
|        | int stopsample() |

| Actuator | int on() |
|          | int off() |

**Flash memory**

| readMemory(int *data, char *buf, int length) |
| writeMemory(int *data, char *buf, int length) |
| erasePage(int page) |
| int lseek(int *address) |
| format() |

**LED**

| ledOn(int led) |
| ledOff(int led) |
| ledToggle(int led) |

**Table 6.1 List of the hardware layer interfaces**

### 6.3.3 Hardware layer instance

Combining the hardware component descriptions and the set of interfaces above, we can create the hardware platform-specific model to characterize a particular WSN mote. For example, we consider the Zolertia Z1 platform [225] which consist of a MSP430 microcontroller, a CC2420 radio chip, a 52-pin expansion connector with I2C and UART, as well as two integrated sensors. The hardware PSM for Z1 platform is presented in Figure B.1, Appendix B.
6.4 Operating system layer

6.4.1 Operating system layer interface

The main mission of the Operating System Layer (OSL) is to map the interface provided by the hardware into higher-level services which are understandable by developers. These services are provided by the set of OSL Interface (OSLI). In this section these interfaces are explored through two popular WSN operating systems: TinyOS 2.x, and Contiki 2.x. The services are grouped into six common categories (refer to Equation 6.2 and Figure 6.4):

- **LED service**: LEDs devices can be used to provide information about the state of a sensor node (for instance, red led on if failure condition).
- **Clock service**: This service controls and synchronizes the operations implemented in the application. It provides different timers to support programming. In addition, the clock service also manages the operating duty cycle of the microcontroller and other devices for the purpose of energy saving.
- **Sensor and Actuator service**: They provide a set of operations to manage the process of sampling the environment through sensors (readings), and reacting to the environment through actuators.
- **Scheduling service**: The scheduling takes decisions about the particular execution of the functionality primitives provided by the OS. Depending on the OS, these primitives can be tasks (in TinyOS) or protothreads (in Contiki). Additionally, for a correct management, other services must be provided, such as blocking or synchronization primitives.
- **Storage service**: Typically, motes include an external flash EEPROM memory chip for storing application data, such as aggregated values, sensor data or logs. This flash memory can be used as a permanent storage device. Therefore, the storage services are necessary to support sensor nodes-specific file systems.
- **Network service**: The network service establishes a network protocol stack for wireless communication between sensor nodes. Different configurations and protocols should be taken into account. In some cases, this group may include the serial communication services to allow exchange data between a node and a gateway device or PC through a serial cable (i.e., RS-232 or USB).

For each one of these functionality groups, the interface provided by TinyOS 2.x and Contiki 2.x has been extensively studied, as shown in the next subsections.
6.4.2 OSL instance for TinyOS 2.x

*LED service:* in TinyOS 2.x, the interfaces for LEDs devices are presented in Table 6.2. The value of the LED 0, 1 and 2 depends on the platform. In general, the LED 0, 1 and 2 are mapped into the red, green and yellow LED respectively.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>async command void led0On()</td>
<td>Leds</td>
<td>Turn on LED 0</td>
</tr>
<tr>
<td>async command void led1On()</td>
<td>Leds</td>
<td>Turn on LED 1</td>
</tr>
<tr>
<td>async command void led2On()</td>
<td>Leds</td>
<td>Turn on LED 2</td>
</tr>
<tr>
<td>async command void led0Off()</td>
<td>Leds</td>
<td>Turn off LED 0</td>
</tr>
<tr>
<td>async command void led1Off()</td>
<td>Leds</td>
<td>Turn off LED 1</td>
</tr>
<tr>
<td>async command void led2Off()</td>
<td>Leds</td>
<td>Turn off LED 2</td>
</tr>
<tr>
<td>async command void led0Toggle()</td>
<td>Leds</td>
<td>Toggle LED 0</td>
</tr>
<tr>
<td>async command void led1Toggle()</td>
<td>Leds</td>
<td>Toggle LED 1</td>
</tr>
<tr>
<td>async command void led2Toggle()</td>
<td>Leds</td>
<td>Toggle LED 2</td>
</tr>
<tr>
<td>async command void set(uint8_t n)</td>
<td>Leds</td>
<td>Read the current LEDs info</td>
</tr>
<tr>
<td>async command void set(uint8_t n)</td>
<td>Leds</td>
<td>Set a value into LEDs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command void startOneShot(uint32_t dt)</td>
<td>Timers</td>
<td>Start a timer in dt time units</td>
</tr>
<tr>
<td>command void startPeriodic(uint32_t dt)</td>
<td>Timers</td>
<td>Periodically starts a timer in every dt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time units</td>
</tr>
<tr>
<td>command void stop()</td>
<td>Timers</td>
<td>Preventing a started timer fired</td>
</tr>
<tr>
<td>command void uint32_t getNext()</td>
<td>Timers</td>
<td>Get current time</td>
</tr>
<tr>
<td>async command void sleep()</td>
<td>MCUSleep</td>
<td>Put microcontroller into a low power</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sleep state</td>
</tr>
<tr>
<td>async command void update()</td>
<td>MCUPower-</td>
<td>Compute the best low power state</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td></td>
</tr>
<tr>
<td>command result_t init()</td>
<td>StdControl</td>
<td>Initialize component and subcomponents</td>
</tr>
<tr>
<td>command result_t start()</td>
<td>StdControl</td>
<td>Start component and subcomponents</td>
</tr>
<tr>
<td>command result_t stop()</td>
<td>StdControl</td>
<td>Stop component and subcomponents</td>
</tr>
</tbody>
</table>

*Clock service:* The component *TimerMilliC* is used to express the timer in milliseconds. There are separated primitives for starting timers including: single (*startOneShot*) or periodic (*startPeriodic*). When the timer expires, the corresponding event is signaled. For power saving management TinyOS 2.x offers several mechanisms. For instance, *MCUSleep* sets the microcontroller to the sleep state, which will be woken up when an interruption arises. For devices power management, the interface *StdControl* is used in combination with *SplitControl* and *AsyncStdControl* interfaces. Table 6.3 shows the interfaces for timer management and energy saving.
6.4 Operating system layer

Sensor and Actuator service: In TinyOS 2.x, the sensor devices and sensor boards are represented through sensor drivers [79]. The drivers are generic components, which can be instantiated more than once, virtualizing the access to the sensor devices and providing interface to the upper layer. Sensor reading is implemented using the common interface Read. This interface has as an argument which is the type of the data produced by the interface (typically a uint16_t value). Table 6.4 shows the interface for sensor and actuator service.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command error_t read()</td>
<td>Read&lt;val_t&gt;</td>
<td>Initiates a reading of sensor data value</td>
</tr>
<tr>
<td>command error_t read(uint32_t usPeriod)</td>
<td>ReadStream&lt;val_t&gt;</td>
<td>Starts to fill buffers by sampling with the specified period</td>
</tr>
<tr>
<td>command error_t postBuffer(val_t*buf, uint16_t count)</td>
<td>ReadStream&lt;val_t&gt;</td>
<td>Passes a buffer to the devices indicating its maximum length</td>
</tr>
<tr>
<td>async command error_t read()</td>
<td>ReadNow&lt;val_t&gt;</td>
<td>Initiates a low-latency reading of small sensor data value</td>
</tr>
</tbody>
</table>

Table 6.4 TinyOS 2.x interface for Sensor and Actuator services

Scheduling service: This service provides a basic management of concurrency at two levels: tasks and events, such as was explained in Chapter 0, Section 2.3. Tasks encapsulate a small amount of processing, and are queued into a FIFO stack of 255 positions. TinyOS 2.x uses the next syntax to define and post tasks, as shown in Table 6.5.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result_t post taskname(void)</td>
<td>Scheduler</td>
<td>Put a task at the end of FIFO</td>
</tr>
<tr>
<td>task void taskname(void)</td>
<td>Scheduler</td>
<td>Define a task</td>
</tr>
</tbody>
</table>

Table 6.5 TinyOS 2.x interface for Scheduling service

Storage service: TinyOS 2.x provides three high-level abstractions for permanent storage service: large objects (i.e., program code received from network), small objects (i.e., configuration data) and logs (i.e., recorded data), which can be stored over partitions called volumes, whose size is configured at compilation time. Depending on the applications requirements, the developer must select a suitable abstraction. For each one of them, an interface is offered as represented in Table 6.6.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command error_t erase()</td>
<td>BlockWrite</td>
<td>Clear all data in the large object</td>
</tr>
<tr>
<td>command error_t write(storage_addr_t.addr, void*buf, storage_len_t len)</td>
<td>BlockWrite</td>
<td>Write data to the large object at a given offset</td>
</tr>
<tr>
<td>command error_t sync()</td>
<td>BlockWrite</td>
<td>Ensure all previous writes are</td>
</tr>
<tr>
<td>Prototype</td>
<td>Interfaces</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>command am_addr_t address()</td>
<td>AMPacket</td>
<td>Return the node's Active Message (AM) address</td>
</tr>
<tr>
<td>command am_addr_t destination(message_t *ams)</td>
<td>AMPacket</td>
<td>Return the AM address of the destination</td>
</tr>
<tr>
<td>command am_addr_t source(message_t *ams)</td>
<td>AMPacket</td>
<td>Return the AM address of the</td>
</tr>
</tbody>
</table>

Table 6.6 TinyOS 2.x interface for Storage services

Network service: The ActiveMessage component is responsible for single-hop wireless communication aspects in TinyOS 2.x. The exchanging message is defined using a structure named message_t. The AMPacket and Packet interfaces are used to access the fields of the packet. Serial communication is implemented using SerialActiveMessageC component. Table 6.7 TinyOS 2.x interface for Network service shows the interfaces for network service.
### 6.4 Operating system layer

<table>
<thead>
<tr>
<th>command</th>
<th>void setDestination(message_t *amsg, am_addr_t addr)</th>
<th>AMPacket</th>
<th>Set the AM address of the destination field</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>void setSource(message_t *amsg, am_addr_t addr)</td>
<td>AMPacket</td>
<td>Set the AM address of the source field</td>
</tr>
<tr>
<td>command</td>
<td>bool isForMe(message_t *amsg)</td>
<td>AMPacket</td>
<td>Return true if the specified message is destined to the mote</td>
</tr>
<tr>
<td>command</td>
<td>am_id_t type(message_t *amsg)</td>
<td>AMPacket</td>
<td>Return the AM type field of the AM packet</td>
</tr>
<tr>
<td>command</td>
<td>void setType(message_t *amsg, am_id_t type)</td>
<td>AMPacket</td>
<td>Set the AM type field of the AM packet</td>
</tr>
<tr>
<td>command</td>
<td>am_group_t group(message_t *amsg)</td>
<td>AMPacket</td>
<td>Return the AM group of the AM packet</td>
</tr>
<tr>
<td>command</td>
<td>void setGroup(message_t *amsg, am_group_t grp)</td>
<td>AMPacket</td>
<td>Set the AM type group of the packet</td>
</tr>
<tr>
<td>command</td>
<td>am_group_t localGroup()</td>
<td>AMPacket</td>
<td>Return the current AM group of this interface</td>
</tr>
<tr>
<td>command</td>
<td>error_t send(am_addr_t addr, message_t *msg, uint8_t len)</td>
<td>AMSend</td>
<td>Send a packet to a specified address</td>
</tr>
<tr>
<td>command</td>
<td>error_t cancel()</td>
<td>AMSend</td>
<td>Cancel a requested transmission</td>
</tr>
<tr>
<td>command</td>
<td>void clear(message_t *msg)</td>
<td>Packet</td>
<td>Clear the packet</td>
</tr>
<tr>
<td>command</td>
<td>uint8_t payloadLength(message_t *msg)</td>
<td>Packet</td>
<td>Return the length of the payload of a packet</td>
</tr>
<tr>
<td>command</td>
<td>void setPayloadLength(message_t *msg, uint8_t len)</td>
<td>Packet</td>
<td>Set a length of the payload field of a packet</td>
</tr>
<tr>
<td>command</td>
<td>uint8_t maxPayloadLength()</td>
<td>Packet</td>
<td>Return the maximum payload length</td>
</tr>
<tr>
<td>command</td>
<td>void* getPayload(message_t *msg, uint8_t len)</td>
<td>Packet</td>
<td>Return a pointer to payload region within a packet</td>
</tr>
<tr>
<td>command</td>
<td>uint8_t payloadLength(message_t *msg)</td>
<td>Receive</td>
<td>Return the length of the payload of the received packet</td>
</tr>
<tr>
<td>command</td>
<td>void* getPayload(message_t *msg, uint8_t len)</td>
<td>Receive</td>
<td>Return a pointer to payload region within a received packet</td>
</tr>
</tbody>
</table>

| Table 6.7 TinyOS 2.x interface for Network services |

Now we can construct the OS platform specific model for the TinyOS 2.x. It can be viewed as a specific instance from the general model shown in Figure 6.4. The OS PSM for TinyOS 2.x is presented in Figure B.2, Appendix B.

### 6.4.3 OSL instance for Contiki 2.x

The Contiki interface is classified following the same criterion as in the previous cases. Because Contiki is developed in C programming language, the interface prototypes and variable structures are declared using C language as well.
**LED service:** Contiki uses keywords (i.e., `LEDS_RED`, `LEDS_GREEN`, `LEDS_YELLOW`) to reference LEDs, and declares a common and shared interface to manage them. The interface for LED devices is presented in Table 6.8.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void leds_on(unsigned char leds)</td>
<td>Turn a LED on</td>
</tr>
<tr>
<td>void leds_off(unsigned char leds)</td>
<td>Turn a LED off</td>
</tr>
<tr>
<td>void leds_toggle(unsigned char leds)</td>
<td>Toggle a LED</td>
</tr>
<tr>
<td>unsigned char leds_get(void)</td>
<td>Read the current LED information</td>
</tr>
<tr>
<td>void leds_arch_init(void)</td>
<td>Initialize LED system</td>
</tr>
<tr>
<td>void leds_arch_set(unsigned char val)</td>
<td>Set a value to LED system</td>
</tr>
</tbody>
</table>

Table 6.8 Contiki interface for LED services

**Clock service:** In Contiki, timers and event management are separated into two different libraries: `timer` and `event timers`. The first one provides the simplest form of timers and is used to check if a time period has passed. The applications need to ask the timers if they have expired. The `event timer` library carries out both tasks: managing timers and posting the events associated to applications. The structure supporting an event timer is named `etimer`. Applications must get a pointer to that structure in order to accomplish different tasks, such as starting and stopping a timer. Table 6.9 shows such interface.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void etimer_set(struct etimer *et, clock_time_t interval)</td>
<td>Set an event timer</td>
</tr>
<tr>
<td>void etimer_stop(struct etimer *et)</td>
<td>Stop a pending event timer</td>
</tr>
<tr>
<td>void etimer_restart(struct etimer *et)</td>
<td>Restart an event timer</td>
</tr>
<tr>
<td>int etimer_expired(struct etimer *et)</td>
<td>Check if an event timer has expired</td>
</tr>
<tr>
<td>void clock_init(void)</td>
<td>Initialize the Clock library</td>
</tr>
<tr>
<td>clock_time_t clock_time(void)</td>
<td>Get current time</td>
</tr>
</tbody>
</table>

Table 6.9 Contiki interface for Clock services

**Sensor and Actuator service:** In Contiki, every sensor device has a specific implementation and is encapsulated into a header file and an implementation file, called `sensorname-sensor.h` and `sensorname-sensor.c` respectively, where `sensorname` is substituted by the sensor in particular such as temperature, light, and so on (i.e., `temperature-sensor.c`, `light-sensor.c`). Depending on the sensor used in the application, the corresponding header file should be included to provide necessary interfaces. In general, the interface offered by the header files is common to the complete set of sensors. Readings from sensors return an int value. The interface for sensor and actuator service is shown in Table 6.10.
6.4 Operating system layer

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void init (void)</td>
<td>Initialize a sensor</td>
</tr>
<tr>
<td>static int irq (void)</td>
<td>Initialize an interrupt associated to the sensor</td>
</tr>
<tr>
<td>static void activate(void)</td>
<td>Activate a sensor</td>
</tr>
<tr>
<td>static void deactivate(void)</td>
<td>Deactivate a sensor</td>
</tr>
<tr>
<td>static void active(void)</td>
<td>Check if a sensor is active</td>
</tr>
<tr>
<td>static int configure(int type, void *c)</td>
<td>Configure a sensor</td>
</tr>
<tr>
<td>static unsigned int value(int type)</td>
<td>Get a sample value from a sensor</td>
</tr>
</tbody>
</table>

Table 6.10 Contiki interface for Sensor and Actuator services

Scheduling service: As explained in the Section 2.3, Chapter 0, protothreads are used to allow the sequential flow control for the Contiki processes. Scheduling services and declaration of protothreads are described in Table 6.11. Note that we focus on specifying the services based on protothreads, in order to present an equivalent execution model to task in TinyOS 2.x.

<table>
<thead>
<tr>
<th>Protothread declaration and synchronization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define PROCESS_BEGIN()</td>
<td>Define the beginning of the process</td>
</tr>
<tr>
<td>#define PROCESS_END()</td>
<td>Define the end of the process</td>
</tr>
<tr>
<td>#define PROCESS_WAIT_EVENT()</td>
<td>Wait for an event to be posted to the process</td>
</tr>
<tr>
<td>#define PROCESS_WAIT_EVENT_UNTIL(c)</td>
<td>Wait for an event to be posted to the process, with an extra condition</td>
</tr>
<tr>
<td>#define PROCESS_YIELD()</td>
<td>Yield the currently running process</td>
</tr>
<tr>
<td>#define PROCESS_YIELD_UNTIL(c)</td>
<td>Yield the currently running process until a condition occur</td>
</tr>
<tr>
<td>#define PROCESS_WAIT_UNTIL(c)</td>
<td>Wait for a condition to occur</td>
</tr>
<tr>
<td>#define PROCESS_WAIT_WHILE(c)</td>
<td>Wait while the c condition occur</td>
</tr>
<tr>
<td>#define PROCESS_EXIT()</td>
<td>Exit the currently running process</td>
</tr>
<tr>
<td>#define PROCESS_PT_SPAWN(pt, thread)</td>
<td>Spawn a protothread from the process</td>
</tr>
<tr>
<td>#define PROCESS_PAUSE()</td>
<td>Pause the process for a short while</td>
</tr>
</tbody>
</table>

Table 6.11 Contiki interface for Scheduling services
Storage service: Contiki provides a file system named Coffee to support permanent storage service. Coffee uses a POSIX-like interface, as shown in Table 6.12.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int cfs_coffee_reserve (const char *name, cfs_offset_t size)</td>
<td>Reserve space for a file</td>
</tr>
<tr>
<td>int cfs_coffee_configure_log (const char *file, unsigned log_size, unsigned log_entry_size)</td>
<td>Configure the on-demand log file</td>
</tr>
<tr>
<td>int cfs_coffee_format (void)</td>
<td>Format the storage area assigned to Coffee</td>
</tr>
<tr>
<td>void * cfs_coffee_get_protected_mem (unsigned *size)</td>
<td>Return a protected area pointer which is used to protect the coffee state during the checkpointing operations.</td>
</tr>
<tr>
<td>CCIF int cfs_open (const char *name, int flags)</td>
<td>Open a file</td>
</tr>
<tr>
<td>CCIF void cfs_close (int fd)</td>
<td>Close an opened file</td>
</tr>
<tr>
<td>CCIF int cfs_read (int fd, void *buf, unsigned int len)</td>
<td>Read data from an opened file</td>
</tr>
<tr>
<td>CCIF int cfs_write (int fd, const void *buf, unsigned int len)</td>
<td>Write data to an opened file</td>
</tr>
<tr>
<td>CCIF cfs_offset_t cfs_seek (int fd, cfs_offset_t offset, int whence)</td>
<td>Seek to a specified position in an opened file</td>
</tr>
<tr>
<td>CCIF int cfs_remove (const char *name)</td>
<td>Remove a file</td>
</tr>
<tr>
<td>CCIF int cfs_opendir (struct cfs_dir *dirp, const char *name)</td>
<td>Open a directory for reading entries</td>
</tr>
<tr>
<td>CCIF int cfs_readdir (struct cfs_dir *dirp, struct cfs_dirent *dirent)</td>
<td>Read a directory entry</td>
</tr>
<tr>
<td>CCIF void cfs_closedir (struct cfs_dir *dirp)</td>
<td>Close a directory opened with cfs_opendir()</td>
</tr>
</tbody>
</table>

Table 6.12 Contiki interface for Storage services

Network service: Contiki implements two communication stacks: uIP and Rime (refer to Section 2.3.3). In the proposed rule-based programming model, we considered two communication protocols provided by Rime stack: abc (broadcast) and uc (unicast). Each protocol has a similar interface composed of three public functions: open and close a connection, and send a packet using that connection. Each protocol manages events (such as receptions or retransmissions) through callbacks or bottom-up signals produced among modules. For serial communication, Contiki offers interfaces to support RS-232 and UART protocol. Table 6.13 depicts the basic Contiki interface for Network service.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern rimeaddr_t rimeaddr_node_addr</td>
<td>Define the Rime address of the node</td>
</tr>
<tr>
<td>void rimebuf_clear(void)</td>
<td>Reset Rime buffer</td>
</tr>
<tr>
<td>int rimebuf_copyfrom(const char *from, uint16_t len)</td>
<td>Copy data to Rime buffer</td>
</tr>
<tr>
<td>void abc_open(struct abc_conn *c, uint16_t channel, const struct abc_callbacks *u)</td>
<td>Setup an abc connection</td>
</tr>
<tr>
<td>int abc_send(struct abc_conn *c)</td>
<td>Send a packet using abc</td>
</tr>
<tr>
<td>void abc_close(struct abc_conn *c)</td>
<td>Close an abc connection</td>
</tr>
</tbody>
</table>
### 6.4 Operating system layer

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void uc_open(struct uc_conn *c, uint16_t channel, const struct uc_callbacks *u)</code></td>
<td>Setup an uc connection</td>
</tr>
<tr>
<td><code>int uc_send(struct uc_conn *c, rimeaddr_t*receiver)</code></td>
<td>Send a packet using uc</td>
</tr>
<tr>
<td><code>void uc_close(struct uc_conn *c)</code></td>
<td>Close an uc connection</td>
</tr>
<tr>
<td><code>void rs232_init(void)</code></td>
<td>Initialize RS-232</td>
</tr>
<tr>
<td><code>void rs232_print(char *cptr)</code></td>
<td>Print a text string using RS-232</td>
</tr>
<tr>
<td><code>void rs232_set_input (int(*)(unsigned char))</code></td>
<td>Set input handler for incoming data on RS-232</td>
</tr>
<tr>
<td><code>void uart1_init(unsigned long ubr)</code></td>
<td>Initialize UART</td>
</tr>
<tr>
<td><code>void uart1_writeb(unsigned char c)</code></td>
<td>Write a byte to UART buffer</td>
</tr>
<tr>
<td><code>uint8_t uart1_active(void)</code></td>
<td>Check the status of UART port.</td>
</tr>
<tr>
<td></td>
<td>Return true if UART is busy</td>
</tr>
</tbody>
</table>

Table 6.13 Contiki interface for Networking services

Now we can construct the OS platform specific model for the Contiki 2.x based on the general model shown in Figure 6.4. Figure B.3, Appendix B, presents the OS PSM for Contiki2.x.
7 Application modeling

The application layer and WSN application modeling are discussed in this chapter. In order to classify and capture WSN applications’ properties, we performed an analysis of WSN application requirements which is focused solely on non-functional requirements and constraints. The analysis result is presented in the first subsection of the chapter. Based on the result, the non-functional properties are characterized and classified in the second subsection. In the last subsection of the chapter, the Domain Specific Language and the Platform Independent Model of WSN application are proposed.

7.1 WSN application requirement analysis

Our aim is to identify the key characteristics that differentiate various applications classes and propose a classification of WSN application requirements, especially non-functional requirements. In order to achieve this goal, first, we surveyed WSN applications to identify specific application classes. Then we extracted key properties from the evaluation criteria of well known applications, routing and MAC protocols.

7.1.1 WSN application classification

The classification of WSN applications has been studied in different works. In [6, 7] the authors classified WSN applications into five categories: military, environment, health, home and industry. In [175] the authors surveyed a range of deployed applications to define a WSN design space representing network designers’ choices, deployment modes, and application requirements. A taxonomy of WSN applications is presented in [139], where the dimensions are the different aspects of the application behaviors such as goal, interaction pattern, mobility, space and time. Literatures [40, 63, 134, 154, 157] analyze some WSN applications that employ new emerging technology such as cloud computing, smart building or internet of things.

In our work, the WSN applications can be classified into two categories: monitoring and tracking. The monitoring applications include indoor/outdoor environmental monitoring, health monitoring, power monitoring, inventory location monitoring, factory and process automation, and seismic and structural monitoring. The tracking applications include tracking objects, animals, humans, and vehicles. Figure 7.1 illustrates the WSN application classification.
7.1 WSN application requirement analysis

7.1.2 WSN application evaluation criteria

In order to identify the comprehensive set of WSN application requirements, we also considered different evaluation criteria which are used in many WSN protocol development studies. We reason that the evaluation criteria should reflect key characteristics of the systems and the applications proposed by the authors. These protocol studies are described in Section 2.2, Chapter 2 of the thesis. In the following, we provide a brief description of the used criteria.

7.1.2.1 Efficiency

Given the strict constraints of WSN, almost all reviewed evaluation strategies include some form of efficiency metric. The efficiency metric can be the cost efficiency (i.e., cost of the sensors), the memory efficiency (i.e., program footprint, storage requirement), or the energy efficiency. The metric for energy efficiency evaluation includes:

- The effective lifetime of the WSN

![WSN application classification diagram](image-url)
• The average energy consumed by the network to deliver a packet from source to destination (i.e., *energy per delivered packet*).

• The average dissipated energy which is the ratio of total energy used per node to number of events detected (i.e., *energy per detected event*).

### 7.1.2.2 Performance

WSN performance can be defined in terms of normalized latency, which is the average time taken for a packet to traverse from the source to the destination within the network (i.e., *latency per delivered packet*). In other applications the term performance may refer to the *reaction time* which is the average time it takes for the sink to receive an update after some change occurs in the network. In routing protocol evaluation, performance may refer to the *control overhead* (the ratio between control and data packets in the network), or the *average route length* in hops.

### 7.1.2.3 Reliability

WSN reliability is defined in terms of packet delivery. Normally, the metric for reliability is the *packet loss ratio*. Other application may employ the *event delivery ratio* which is the ratio of the number of distinct event messages received by the sink to the number originally sent by the source. Additionally, the ratio of overall packets to the number of queries injected into the network (the *transmissions to query ratio*) can also be used to evaluate the reliability.

### 7.1.2.4 Other criteria

A number of other evaluation criteria, which concern deployment and design related issues, are mentioned as follows:

- **Security**
- **Ease of deployment**
- **Maintainability**
- **Reprogramming ability**
- **Scalability**

### 7.2 Non-functional properties classification

Our purpose is to classify WSN non-functional properties such that we can choose proper measuring and optimization techniques based on the classification. Some non-functional properties can be described only qualitatively, whereas other properties can be represented with metric-based values. It means that we cannot use the same measuring and optimization technique for all properties. For example, we cannot
compute which solution is the best in term of scalability or easy of development, because we usually have no metric to obtain quantifiable measures. But, we can determine which solution has the smallest footprint or lowest latency. Hence, we classify non-functional properties with respect to our ability to measure them and which operations are valid for the measurement.

We classify non-functional properties into three different classes: qualitative properties, inferable properties and measureable properties. It is important to note that the categorization of a particular non-functional property is not general and depends on the application scenario and the developer’s viewpoint. This means that the same property can be in different classes for different scenario. For instance, in a protocol using hash functions to encrypt the data transmission, security may be measured via the bit-length of the hash values resulting in quantifiable measures. In another scenario, security can only be qualitatively specified (i.e., with weak, medium and strong secure).

**Qualitative non-functional properties**

There are non-functional properties that can only be described qualitatively using an ordinal scale because there is no metric from which we can retrieve quantifiable measures. The ordinal scale must be assigned manually by the developer based on their experience of the application domain. Common representatives of this class are: security, easy of deployment, and scalability.

**Inferable non-functional properties**

This category contains properties that can be measured or inferred in the design phase without executing the application program. Examples of this class are cost of the hardware devices, footprint of the program, or maintainability (which can be measured to some degree with code metrics such as total lines of code and cyclomatic complexity). A measurement allows us to annotate each implementation unit with a specific value (i.e., cost of a sensor device) and to compute a value for a solution. To compute a value for a concrete solution, the developer defines an aggregation function, such as addition or maximum which is used to aggregate the values for each implementation unit. For example, we can define addition as aggregation function for cost or footprint.

**Measureable non-functional properties**

Some properties can only be measured by executing the application. They require the highest measurement effort, because we have to execute and compute the feedback value for a concrete configuration scenario. Common representatives of this class are:
Table 7.1 summarizes the discussion presented in this thesis.

<table>
<thead>
<tr>
<th>Non-functional properties</th>
<th>Requirement descriptions</th>
<th>Class</th>
<th>Measuring technique</th>
<th>Value range</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifetime</td>
<td>Energy efficiency</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>Hours, days, months, years</td>
</tr>
<tr>
<td>Energy delivered packet</td>
<td>Energy efficiency</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>Jpacket$^{-1}$</td>
</tr>
<tr>
<td>Energy per detected event</td>
<td>Energy efficiency</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>Jpacket$^{-1}$ event$^{-1}$</td>
</tr>
<tr>
<td>Latency delivered packet</td>
<td>Performance requirement</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>packet$^{-1}$s</td>
</tr>
<tr>
<td>Reaction time</td>
<td>Performance requirement</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>S</td>
</tr>
<tr>
<td>Control overhead</td>
<td>Performance requirement</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>Unitless</td>
</tr>
<tr>
<td>Average route length</td>
<td>Performance requirement</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>Hops</td>
</tr>
<tr>
<td>Packet loss ratio</td>
<td>Reliability requirement</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>[0,1]</td>
<td>Unitless</td>
</tr>
<tr>
<td>Event delivery ratio</td>
<td>Reliability requirement</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>Unitless</td>
</tr>
<tr>
<td>Transmissions to query ratio</td>
<td>Reliability requirement</td>
<td>Measurable</td>
<td>Using simulator</td>
<td>(0,∞)</td>
<td>Unitless</td>
</tr>
<tr>
<td>Hardware cost</td>
<td>Cost efficiency</td>
<td>Inferable</td>
<td>Compute the in design phase</td>
<td>(0,∞)</td>
<td>USD</td>
</tr>
<tr>
<td>Footprint</td>
<td>Memory efficiency</td>
<td>Inferable</td>
<td>Compute the in design phase</td>
<td>(0,∞)</td>
<td>Bytes, Kbytes</td>
</tr>
<tr>
<td>Storage requirement</td>
<td>Memory efficiency</td>
<td>Inferable</td>
<td>Compute the in design phase</td>
<td>(0,∞)</td>
<td>Bytes</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Design requirement</td>
<td>Inferable</td>
<td>Compute the in design phase</td>
<td>(0,∞)</td>
<td>Lines of code</td>
</tr>
<tr>
<td>Security</td>
<td>Design requirement</td>
<td>Qualitative</td>
<td>Defined by developer</td>
<td>[Week, Medium, Strong]</td>
<td></td>
</tr>
<tr>
<td>Easy of deployment</td>
<td>Design requirement</td>
<td>Qualitative</td>
<td>Defined by developer</td>
<td>[Easy, Medium, Hard]</td>
<td></td>
</tr>
<tr>
<td>Reprogramming ability</td>
<td>Design requirement</td>
<td>Qualitative</td>
<td>Defined by developer</td>
<td>[Yes, No]</td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td>Design requirement</td>
<td>Qualitative</td>
<td>Defined by developer</td>
<td>[Easy, Medium, Hard]</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1 Summarize of WSN non-functional properties
7.3 Application layer

7.3.1 Domain Specific Language

We design a DSL that takes into account all the concepts mentioned above, as shown in Figure 7.2.

The *Message* class is used to represent the message structure, as discussed in Section 5.3, and each message is encoded using *(attribute, value)* pairs. The *type* describes the message and the attached content. Additional parameters of the message can be added later based on the application’s requirements.
Chapter 7  Application modeling

The *Sensor* class of the DSL meta-model represents sensor devices that are used in the motes. An application may consist of one or more sensor devices. Attribute *type* of the *Sensor* class represents sensor type. The class *SamplingRate* is used to express the sampling interval. This class is categorized as an *AdjustableParameter* class which is used to represent the adjustable parameters for the optimization purpose. After each sampling cycle the collected data will be packed in to a message, and this message will be automatically added to the message buffer.

The message buffer is represented by using *MsgBuffer* class. The class *BufferLength*, which is also an adjustable parameter, defines the maximum length of the buffer. An application has only one message buffer. The message in the buffer comes from two different sources: from the *Sensor* class and from the *Receiver* class. The annotation <<create>> represents this relationship between these classes.

The *CommInterface* element expresses the communication abilities of the network’s nodes. An application may contain one or more *CommInterface* classes such as *RADIO* or *SERIAL* according to its specific requirements. Each *CommInterface* includes a *Receiver* and a *Sender*. Every time a message comes to the *Receiver*, this message is automatically added to the message buffer. The *Sender* employs a message queue, which is represented by the *SendingQueue* class, to support sending task, the *SendingQueueLength* depicts the maximum length of the sending queue. *SendingQueueLength* is an adjustable parameter.

A *RuleSet* class and a *Rule* class are also included in the DSL. The *RuleSet* class has a *Name* attribute to define the name of the rule set. Each *RuleSet* includes an *ExecutionTime* to refer the execution interval of the rules in the rule set. Similar to *SamplingRate*, the *ExecutionTime* is an adjustable parameter. Three elements including *SelectClause*, *ProcessingClause* and *ActionClause* form a rule in the rule set. Because a rule may contain many actions, the *next* association is included in the *ActionClause* class. The rule’s in-network processing functions and actions are defined by using the attribute *FunctionName* and *ActionName* in the corresponding class. It is worth noting that at this level of abstraction the implementation of the rule interpreter, the in-network processing functions and the actions are abstracted from the developer. The code performs the rule’s actions and rule translation will be added later during the transformation process.

The *AdjustableParameter* class represents the customizable parameters of the application. The developer must define corresponding values for every attributes of this class including *Name*, *DefaultValue*, *Minvalue* and *Maxvalue*. These definitions will be used later.
in the optimization process, where a proper value of each parameter is chosen from the value range $[\text{Minvalue}, \text{Maxvalue}]$.

Finally, the non-functional properties are represented by using the $\text{NRProperty}$ class. An application may include many non-functional properties of different types such as $\text{Qualitative}$, $\text{Measurable}$ or $\text{Inferable}$. The developer should define the $\text{ObjectiveFunction}$ of the non-functional requirement represented by $\text{NRRequirement}$ class.

### 7.3.2 Graphical development framework

We developed a graphical development framework for composing applications at the application layer. The framework supports describing WSN applications based on the defined DSL, but the language elements (i.e., sensor, buffer, rules) are substituted with a graphical notation. The benefit of using a visual tool is twofold:

- To guide the development process through an Integrated Development Environment (IDE) in order to avoid programming errors
- To facilitate high-level programming through a graphical interface.

Clearly, the graphical framework will result in less effort and shorter development time of WSN applications. Among various MDD supporting tools, the Eclipse Modeling Project [164] is an open source platform which unifies the modeling related frameworks, tools and standard implementations. This project includes:

- **Eclipse Modeling Framework (EMF)**: EMF is a modeling framework and code generation facility for specifying metamodels and managing model instances. EMF includes its own metametamodel called Ecore which is used for describing and runtime support for other models.
- **Graphical Editing Framework (GEF)**: GEF allows developers to create a rich graphical editor from an existing application model. GEF employs a Model-View-Controller (MVC) architecture which enables simple changes to be applied to the model from the view. GEF provides the graphical support required for building a diagram editor on top of the EMF framework.
- **Graphical Modeling Framework (GMF)**: GMF provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. Actually, GMF functions like a generative bridge between EMF and GEF. GMF employs MOFScript to provide support for model to text transformations.
In general, one can generate graphical editors for DSLs by using either EMF/GEF or GMF. Since GMF supports creating a graphical definition and mapping definition to the chosen DSL, we selected GMF to develop the graphical framework. Figure 7.3 illustrates the main components, the models and the development process:

- The DSL metamodel is developed by defining an Ecore model with EMF Editor. The diagram definition (.ecore_diagram) is maintained separately from the metamodel in GMF. In addition, EMF Generator Model (.genmodel) should be created from Ecore model using an EMF wizard.
- Graphical Definition (.gmfgraph) is developed by using the GMF Graph Model wizard to generate a graphical definition model from the existing Ecore model. Graphical definition model contains information related to the graphical elements (figures, nodes, compartments, links, etc.) that will appear on the produced graphical framework.
- Tooling Definition (.gmftool) is developed by using the GMF Tool Model wizard in order to provide tooling support for the graphical definition including palettes, menus, and toolbars.
- Mapping Model (.gmfmap) is developed using the GMF Map Model wizard. A separate mapping model is used to link the graphical and tooling definitions to the selected DSL metamodel.
- Once the appropriate mappings are developed, a Generator Model (.gmfgen) - similar to EMF .genmodel is created from the graphical definition, tooling definition, and mapping models to allow specifying the implementation details for the code generation phase.
- Finally, the graphical framework is generated and then tested in a new Eclipse application at runtime. The framework provides the user an IDE to work with the DSL elements.
Following these steps, we developed the graphical framework for WSN application development using the proposed DSL. Firstly, we defined the DSL metamodel (Figure 7.2) based on the Ecore meta-model. Secondly, we created the necessary graphical definition models and tooling definition models. This process consists of selecting the suitable icons identifying the DSL’s elements, and arranging them on a palette, and toolbar, which contains the visual representation of the application. Then the mapping models and generator models are created, and the framework is generated as an Eclipse plugin. Table 7.2 describes the graphical notation of our DSL.

<table>
<thead>
<tr>
<th>DSL elements</th>
<th>Descriptions</th>
<th>Graphical notation</th>
<th>Sharps / Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application class</td>
<td>Resizeable rectangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The static text Application=Application.Name is placed on the top left corner</td>
<td></td>
</tr>
<tr>
<td>AdjustableParameter class</td>
<td>Static text</td>
<td>Adjustable Parameter=(Name, DefaultValue, MinValue, MaxValue)</td>
<td></td>
</tr>
<tr>
<td>Sensor class</td>
<td>Icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CommInterface class</td>
<td>Icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receiver class</td>
<td>Icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sender class</td>
<td>Icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MsgBuffer class</td>
<td>Icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SendingQueue class</td>
<td>Icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message class</td>
<td>Icon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MessagePair class</td>
<td>Static text</td>
<td>{MessagePair.attribute, MessagePair.value}</td>
<td></td>
</tr>
<tr>
<td>RuleSet class</td>
<td>Resizeable Rectangle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The static text Rule Set=RuleSet.Name is placed on the top left corner.</td>
<td></td>
</tr>
</tbody>
</table>
7.3.3 An example of WSN application PIM model

In this section we demonstrate how to apply our DSL to create a PIM model for a simple application. In our example a sensor network is used to measure the temperature in an area. Each node in the network is equipped with a sensor to periodically sense the local temperature after a time $T$ (for example $T=250\text{ms}$ - four times per second). The temperature data is forwarded along the network to a base station. Data aggregation is performed in every node to reduce the number of transmitted messages. Additionally, critical temperature values are observed and an alert message is sent to the base station if a threshold has been exceeded.

![Diagram of PIM model](image)

As shown in the Figure 7.4, all elements in the DSL meta-model have a corresponding graphical representation such as an icon or a colored box. The developer assigns...
necessary values for the adjustable parameters like SamplingRate or BufferLength to define the sensing cycle and maximum length of the message buffer. The default value and the value range of each parameter are also defined by setting the value of the Defaultvalue, Minvalue and Maxvalue attributes. For example, the text \((\text{SamplingRate1, 250, 1, 1000})\) in the PIM means the adjustable parameter SamplingRate of the first sensor (the Temperature Sensor) has the default value 250 ms, and a value range from 1ms to 1000ms. This range will be used later in the optimization process.

In order to describe the behaviors of the sensor node, a rule set that consists of three members is predefined. After a time \(T_s\) (for example \(T_s=2s\)) the rule interpreter is triggered to apply the set of rules to all message in the buffer. The first rule is used to send an alert message to the base station when the threshold temperature is exceeded. In the second rule, an in-network processing function \((\text{Average(temperature)})\) is used to aggregate data. The ActionClause has two actions, the first one simply forwards the messages to the next hop in the network and the second one discards all the processed messages. The third rule is used to continue forwarding the alert message to the base station. In this case, a simple flooding routing strategy is used. When a node receives an alert message from its neighbor, the node will continue broadcasting the message.

In a temperature monitoring application, developers often care about the reliability and energy efficiency of the system. Therefore, non-functional properties such as energy per delivered packet \((NFP1)\) and reaction time \((NFP2)\) could also be defined. The developer may want to optimize the application in term of energy efficiency and reliability, so he may define the non-functional constrains as \(\text{Min}(NFP1) \&\& \text{Min}(NFP2)\). This condition will be used later in the optimization process. Listing B.2, Software ArchitectureAppendix B contains the XML file named application.xml, which is used to describe the application using the Ecore metamodel.
8 Transformation and code generation

In this chapter the transformation and code generation are discussed. These activities are performed in the Operating System Abstraction Layer (OAL) of the proposed software architecture. In general, the OAL should take as the input a PIM model represented a WSN application, and automatically generate the equivalent code for the selected target platform (operating system and hardware). The transformation process is separated in three different steps: the first one converts the PIM application into an intermediate program code which is constructed using a set of OS independent building blocks and functions. The second parses the intermediate code and generate the equivalent code for a target operating system. In the final step, the binary code for specific motes is created using the OS-specific compiler. The transformation process and the steps are discussed in detail in the following subsections of the chapter.

8.1 Overview of the transformation process

Figure 8.1 depicts the transformation process which consists of three steps: mapping, translation and compiling. The steps are listed below:

- **Mapping**: The input application PIM (file application.xml), is mapped to the intermediate program code consisting of a header file (application.h) and a C program file (application.c). The purpose of the mapping process is to implement the concept of rule-based programming using C language building blocks and functions. During the mapping process, a code template file (template.c) which
8.1 Overview of the transformation process

contains reusable code snippets is used in combination with an XML parser to parse the PIM and generate the C program file. The set of necessary code blocks and functions are presented in the next subsection of the chapter.

- **Translation**: This process takes as input the intermediate program code files (application.c, application.h), and maps them into operating system-specific services defined by the OSLI (refer to Section 6.4.1), generating the operating system-specific application. In other words, this process performs the translation function $f_T$ previously described in Section 6.2.

- **Compiling**: In this step the OS specific program will be compiled using the supported compiler. With a selected hardware platform, the makefiles are created to specify how to derive the target binary code. Depending on the OS, different compilers are used, for example, ncc compiler for TinyOS program and gcc compiler for Contiki program.

Extending the basic design in Figure 6.5, the instance component diagram of the Operating System Abstraction Layer (OAL) is depicted in Figure 8.2.

---

**Figure 8.2 Instance component diagram of the Operating System Abstraction Layer**
8.2 Rule interpreter component

The purpose of the rule interpreter component is to map a rule-based program into a C program. This mapping process will allow the developer to:

- Separate the application logic (WSN node behavior) from the concrete application implementation.
- Focus more on design and represent node behavior using a declarative language.
- Reuse the mapped C building blocks and functions in other applications.

A configuration file in XML format is used to represent user defined information such as the workspace location, model files, code files, target platform, deployment and simulation configuration, and optimization configuration. The XML file is dynamically updated at development time when developers specify the information. An example of the XML configuration file is presented in Listing 8.1.

### A XML file to describe the application configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<CONFIGURATION>
  <DEFINITION>
    <AppName>Application Name</AppName>
    <Path>/workspace/wns.org.eclipse.dsl.project</Path>
  </DEFINITION>
  <INDEPENDENTAPP>
    <PimApp>./model/application.xml</PimApp>
    <CAppTemplate>./src/template.c</CAppTemplate>
    <CApp>./src/application.c</CApp>
    <CAppHeader>./src/application.h</CAppHeader>
  </INDEPENDENTAPP>
  <TARGET>
    <OperatingSystem name="TinyOS" version="2.x" compiler="ncc" path="/opt/tinyos-2.x" />
    <OsPsm>./model/tinyos.xml</OsPsm>
    <OsCode>./src/TinyOs</OsCode>
    <Platform name="Z1" provider="Zolertia" />
    <HardwarePsm>./model/z1.xml</HardwarePsm>
    <Sensorboard></Sensorboard>
    <BinaryCode>./src/TinyOs</BinaryCode>
  </TARGET>
  <DEPLOYMENT>
  
  </DEPLOYMENT>
  <OPTIMIZATION>
  
  </OPTIMIZATION>
</CONFIGURATION>
```

Listing 8.1 An example of application configuration file

In order to facilitate mapping process, the design of the C building blocks and function library must be taken carefully into account. On one hand, they must be sufficient for
implementing the elements of rule-based programing model such as buffer, sending queue, rule set and individual rule, in-network processing and other actions. On the other hand, they must be designed with the aim to standardize the interface of the two WSN operating system considered, presented in Section 6.4. The function library and the building blocks are presented respectively in the following subsections.

8.2.1 The function library

The function set includes basic operations for implementing the rule-based application. The set is also classified according to the six groups of functionalities provided by the operating system layer including: LED service, clock service, sensor and actuator service, scheduling service, storage service, and networking service. In addition, a group containing utility functions is added in order to support queue managing and random generator. Table 8.1 lists the function library. Note that the function set does not preclude further extensions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LED group</strong></td>
<td></td>
</tr>
<tr>
<td>void led_on(uint8_t led)</td>
<td>Turn on the led device specified by the led argument</td>
</tr>
<tr>
<td>void led_off(uint8_t led)</td>
<td>Turn off the led device specified by the led argument</td>
</tr>
<tr>
<td>void led_toggle(uint8_t led)</td>
<td>Toggle the led device specified by the led argument</td>
</tr>
<tr>
<td><strong>Clock and Timer group</strong></td>
<td></td>
</tr>
<tr>
<td>uint8_t timer_start(uint8_t num, uint8_t scale, uint8_t frequency)</td>
<td>This function is used to set a timer event which will be signaled after a given time. It returns a non-zero timer handle if the operation had success. This handle must be used in subsequent operations on the same timer.</td>
</tr>
<tr>
<td>uint8_t timer_stop(uint8_t timer)</td>
<td>This function stops a timer previously initiated by the timer_start function</td>
</tr>
<tr>
<td>uint8_t timer_restart(uint8_t timer)</td>
<td>This function restarts a timer previously initiated by the timer_start function. The timer is restarted from the current point in time, and it keeps the same configuration.</td>
</tr>
<tr>
<td>uint32_t get_time()</td>
<td>This function returns current time value.</td>
</tr>
<tr>
<td>uint8_t device_on(uint8_t deviceid)</td>
<td>Power on a device specified by the deviceid parameter</td>
</tr>
<tr>
<td>uint8_t device_off(uint8_t deviceid)</td>
<td>Power off a device specified by the deviceid parameter</td>
</tr>
<tr>
<td><strong>Sensor and Actuator group</strong></td>
<td></td>
</tr>
<tr>
<td>uint8_t sensor_init(uint8_t sensor)</td>
<td>Initialize a sensor. This function returns a zero value if the sensor was successfully initialized. Non zero is returned in case of failure.</td>
</tr>
<tr>
<td>uint8_t sensor_read(uint8_t sensor, uint16_t data)</td>
<td>Initiate the reading on a device sensor. This operation includes the sampling over a sensor and the ADC conversion of data. The data argument is loaded with the resulting 16-bit value.</td>
</tr>
</tbody>
</table>
### Chapter 8  Transformation and code generation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8_t sensor_stop(uint8_t sensor)</code></td>
<td>Stop a sensor. This function returns a zero value if the sensor was successfully initialized. Non zero is returned in case of failure.</td>
</tr>
</tbody>
</table>

### Tasks and Scheduling group

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8_t task_create(char *name, void * (*start_routine)(void *), void *args)</code></td>
<td>Schedule a new process/task to be laterly executed. The name argument is the logical name of the process. The start_routine is the function to be executed by the process. The args is a pointer to a structure including the arguments for the process. A process/task is mapped to the equivalent execution entity in the underlying operating system, for which the application will be translated.</td>
</tr>
<tr>
<td><code>uint8_t task_current()</code></td>
<td>Obtain the metadata of the current task</td>
</tr>
<tr>
<td><code>uint8_t task_exit()</code></td>
<td>Remove a task from the task queue</td>
</tr>
<tr>
<td><code>uint8_t task_getid()</code></td>
<td>Obtain the task identifier</td>
</tr>
<tr>
<td><code>uint8_t task_signal(uint8_t event, uint8_t deviceid, signalhandler_t handler)</code></td>
<td>Establish a handler event. This function installs a new handler event for the event identified by event argument. The event handler is established by the third argument called handler, which specifies the function name to be executed when the signal is received. The second argument is the identifier of the device which receives the handler.</td>
</tr>
</tbody>
</table>

### Storage group

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8_t fs_open(char *filename, uint8_t mode)</code></td>
<td>Open a file. This function returns a non-zero file handle if the operation had success. This handle must be used in subsequent operations on the same file.</td>
</tr>
<tr>
<td><code>uint8_t fs_close(uint8_t file)</code></td>
<td>Close a file specified by the file handle</td>
</tr>
<tr>
<td><code>int8_t fs_write(uint8_t file, char *buffer, uint8_t length)</code></td>
<td>Append data to an opened file</td>
</tr>
<tr>
<td><code>uint8_t fs_read(uint8_t file, char *buffer, uint8_t length)</code></td>
<td>Read data from an opened file</td>
</tr>
<tr>
<td><code>uint8_t fs_rename(uint8_t file, char *newname)</code></td>
<td>Rename an existing file with a new name</td>
</tr>
<tr>
<td><code>uint32_t fs_lseek(uint8_t file, uint32_t *ptr)</code></td>
<td>Seek to a specified position in an opened file</td>
</tr>
<tr>
<td><code>uint8_t fs_delete(uint8_t file)</code></td>
<td>Delete a file</td>
</tr>
</tbody>
</table>

### Networking group

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint8_t net_send(uint8_t address, void *data, uint8_t length)</code></td>
<td>This function sends a message to a specified address. The address can be: the local address of a node, the broadcast address (0xFF), or the address of the serial port.</td>
</tr>
<tr>
<td><code>uint8_t net_getId()</code></td>
<td>Get the local address, typically the node ID</td>
</tr>
<tr>
<td><code>uint8_t net_getIdBroadcast()</code></td>
<td>Get the broadcast address, typically 0xFF</td>
</tr>
<tr>
<td><code>uint8_t net_getIdSerial()</code></td>
<td>Get the serial address of the node, typically 0x7E</td>
</tr>
</tbody>
</table>
8.2.2 The building blocks, parser and template file

Beside the function library, it is necessary to define a set of basic blocks of the program. In our work the programming blocks and the corresponding constructs are defined below:

- The main block is represented by the main keyword. This block represents the starting point of the programs. This function must be unique. Depending on the target OS, different specific code will be generated for this initial point.
- Functions are represented by the POSIX-style function prototype. Functions can be considered as high-level tasks, which are created using the task_create function described above, to encapsulate a particular functionality within a program.
- Events are represented in a similar way to functions, but differ from functions in that events encapsulate the code for the event handlers written within programs. The function task_signal is used to connect the signal with the function name to execute when an event occurs.

During the mapping process, a code template file (template.c) which contains reusable code snippets is used in combination with an XML parser to parse the PIM and generate the C program file.

8.3 OS translator component

This component is intended to perform the translation function $f_T$ previously described in Section 6.2. It takes as input the intermediate program code files (application.c, application.h), and maps them into operating system-specific services defined by the OSLI (refer to Section 6.4.1), generating the operating system-specific application.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t queue_init(uint8_t queue_size)</td>
<td>Initialize a queue. The argument queue_size is the maximum queue length</td>
</tr>
<tr>
<td>uint8_t queue_isEmpty(uint8_t queueid)</td>
<td>Return 1 if the queue is empty. Otherwise zero is returned</td>
</tr>
<tr>
<td>uint8_t queue_clear(uint8_t queueid)</td>
<td>Delete all items in a queue.</td>
</tr>
<tr>
<td>uint8_t queue_isFull(uint8_t queueid)</td>
<td>Return 1 if the queue length reaches the maximum value. Otherwise zero is returned</td>
</tr>
<tr>
<td>uint8_t queue_enqueue(uint8_t queueid, msg_t item)</td>
<td>Add an item to a queue. Typically, an item is a message which is defined using msg_t structure</td>
</tr>
<tr>
<td>msg_t queue_dequeue(uint8_t queueid)</td>
<td>Get an item from a queue.</td>
</tr>
</tbody>
</table>

Table 8.1 List of C function library
The component is implemented using the Perl scripting language. A script file is created for each OS like Contiky or TinyOS to generate functionally equivalent application in the corresponding native code. These scripts will parse the input code and replace each primitive block with the platform-specific code. The process consists of main block translation, function translation and event mapping.

### 8.3.1 Main block translation

The equivalent code for the main block is generated in different ways depending on the target OS. In TinyOS the application is split in two files:

- The configuration file identifies the components used by the application and how they are related (wirings).
- The implementation file contains the code for the components’ functionality.

In order to create TinyOS native code, the input code must be parsed to figure out all required components, interfaces, and wirings for the configuration file. Then the equivalent starting point is created by including the interface `Boot` and the event `Booted` into the implementation file.

In case of Contiki, the application is mapped to one native code file. This file contains several functions (protothreads) that are enclosed by the `PROCESS_THREAD` and `PROCESS_END` macros. The `AUTOSTART_PROCESSES` macro is used to indicate the execution order of the defined protothreads. Therefore, the main block must be the first protothread declared in this list. Table 8.2 demonstrates the mapping from the main block to TinyOS and Contiki native code.

<table>
<thead>
<tr>
<th>Primitive block</th>
<th>TinyOS 2.x code</th>
<th>Contiki code</th>
</tr>
</thead>
<tbody>
<tr>
<td>main block</td>
<td>//Configuration file: applicationAppC.nc configuration applicationAppC {</td>
<td>PROCESS(main,&quot;main&quot;);</td>
</tr>
<tr>
<td></td>
<td>} implementation {</td>
<td>AUTOSTART_PROCESS(&amp;main);</td>
</tr>
<tr>
<td></td>
<td>//Component list components MainC; components applicationC;</td>
<td>PROCESS_THREAD(main, ev, data) {</td>
</tr>
<tr>
<td></td>
<td>... other components ...</td>
<td>PROCESS_EXITHANDLER(goto exit);</td>
</tr>
<tr>
<td></td>
<td>//Wiring applicationC.Boot -&gt; MainC.Boot;</td>
<td>PROCESS_BEGIN();</td>
</tr>
<tr>
<td></td>
<td>... other wiring ...</td>
<td>... main block code ...</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>... other implementation ...</td>
</tr>
<tr>
<td></td>
<td>//Implementation file: applicationC.nc module applicationC {</td>
<td>exit:</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>PROCESS_END();</td>
</tr>
</tbody>
</table>
8.3 OS translator component

```c
uses interface Boot;
... other interface ...
}
implementation {
  event void Boot.booted() {
    ... main block code ...
  }
  ... other implementation ...
}
```

Table 8.2 Main block translation to native code in TinyOS 2.x and Contiki

8.3.2 Functions translation

Functions are execution entities in the input program. They must be translated into the equivalent OS-specific code including the configuration code (wiring) and the implementation code. The translation uses regular expressions and patterns listed in Table 8.3. The patterns that are showed in the first column of the table are the functions in the input program. During the translation process, any input code’s statement that matches a pattern will be substituted by the native code of the selected OS in the second or third column of the table. For example, if the target OS is TinyOS 2.x and a statement in the input file matches the pattern `led_on(LED_RED)`, the statement will be replaced with the native code `call Leds.led0On()`.

The symbol “*” can be used in a pattern. In this case, whatever character or string would match this “*”. For instance, the pattern `timer_start("",SECOND,ONESHOT)` is used to find a statement that creates a timer in the input code. The first argument indicates the interval. The second argument expresses the time scale, and the third indicates the frequency. The “*” in the pattern means any character or string can match the first argument, and the value is remembered by the translation module. When substituting, this value can be accessed using the notation “\n”, when n is the order of the argument in the function.

An user-defined function created using `task_create()` will be mapped to a task in TinyOS or a protothread in Contiki. The function prototype indicates the starting point and the location of the execution code. The function’s arguments are processed in the same way as in the function library.

<table>
<thead>
<tr>
<th>Primitive block</th>
<th>TinyOS 2.x code</th>
<th>Contiki code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>led_on(LED_RED)</code></td>
<td><code>call Leds.led0On()</code></td>
<td><code>leds_on(LED_RED)</code></td>
</tr>
<tr>
<td><code>led_on(LED_GREEN)</code></td>
<td><code>call Leds.led10n()</code></td>
<td><code>leds_on(LED_GREEN)</code></td>
</tr>
<tr>
<td><code>led_off(LED_RED)</code></td>
<td><code>call Leds.led0Off()</code></td>
<td><code>leds_off(LED_RED)</code></td>
</tr>
<tr>
<td><code>led_off(LED_GREEN)</code></td>
<td><code>call Leds.led10ff()</code></td>
<td><code>leds_off(LED_GREEN)</code></td>
</tr>
<tr>
<td>Call to Function</td>
<td>Call to Timer Function</td>
<td>Event Handling</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>timer_start(*, SECOND, ONESHOT)</td>
<td>call Timer0.startOneShot(\1 * 1000)</td>
<td>etimer_set(&amp;tX, \1 * CLOCK_CONF_SECOND)</td>
</tr>
<tr>
<td>timer_start(*, SECOND, REPEAT)</td>
<td>call Timer0.startPeriodic(\1 * 1000)</td>
<td>etimer_set(&amp;tX, \1 * CLOCK_CONF_SECOND); etimer_reset(&amp;tX)</td>
</tr>
<tr>
<td>timer_start(*, MILLISECOND, ONESHOT)</td>
<td>call Timer0.startOneShot(\1)</td>
<td>etimer_set(&amp;tX, \1 * CLOCK_CONF_SECOND/ 1000)</td>
</tr>
<tr>
<td>timer_start(*, MILLISECOND, REPEAT)</td>
<td>call Timer0.startPeriodic(\1)</td>
<td>etimer_set(&amp;tX, \1 * CLOCK_CONF_SECOND/ 1000); etimer_reset(&amp;tX)</td>
</tr>
<tr>
<td>timer_restart(*)</td>
<td>call Timer0.stop(); call Timer0.startOneShot(\1)</td>
<td>etimer_reset(&amp;tX)</td>
</tr>
<tr>
<td>timer_stop(*)</td>
<td>call Timer0.stop();</td>
<td>etimer_expired(&amp;tX)</td>
</tr>
<tr>
<td>sensor_init(TEMP)</td>
<td></td>
<td>temperature_sensor.active()</td>
</tr>
<tr>
<td>sensor_read(TEMP)</td>
<td>call TempSensor.read()</td>
<td>data=temperature_sensor.value(0)</td>
</tr>
<tr>
<td>sensor_stop(TEMP)</td>
<td></td>
<td>temperature_sensor.deactive()</td>
</tr>
<tr>
<td>sensor_init(LIGHT)</td>
<td></td>
<td>light_sensor.active()</td>
</tr>
<tr>
<td>sensor_read(LIGHT)</td>
<td>call LightSensor.read()</td>
<td>data= light_sensor.value(0)</td>
</tr>
<tr>
<td>sensor_stop(LIGHT)</td>
<td></td>
<td>light_sensor.deactive()</td>
</tr>
<tr>
<td>sensor_init(ACCEL)</td>
<td>Call AccelSensor.read()</td>
<td>accel_sensor.active()</td>
</tr>
<tr>
<td>sensor_read(ACCEL)</td>
<td></td>
<td>data= accel_sensor.value(0)</td>
</tr>
<tr>
<td>sensor_stop(ACCEL)</td>
<td></td>
<td>accel_sensor.deactive()</td>
</tr>
<tr>
<td>task_create(<em>,<strong>,</strong></em>)</td>
<td>uint_t param; param=\3; post \1(); task void \1() { \2; }</td>
<td>PROCESS_THREAD(\1.ev, \3) { PROCESS_BEGIN(); \2; PROCESS_END(); }</td>
</tr>
<tr>
<td>task_current()</td>
<td></td>
<td>PROCESS_CURRENT()</td>
</tr>
<tr>
<td>task_exit()</td>
<td></td>
<td>PROCESS_EXIT()</td>
</tr>
<tr>
<td>task_signal(IO_READDONE, <strong>,</strong>*)</td>
<td>event void \2.readDone(error_t err, uint16_t val) { \3; }</td>
<td>PROCESS_WAIT_EVENT(); if (ev==event_data_ready) { \3; }</td>
</tr>
<tr>
<td>task_signal(IO_WRITEDONE, <strong>,</strong>*)</td>
<td>event void \2.writeDone(error_t err) { \3; }</td>
<td>PROCESS_WAIT_EVENT(); if (ev==event_data_ready) { \3; }</td>
</tr>
<tr>
<td>task_signal(TIMER_FIRED, *, **)</td>
<td>event void \2.fired() { \3; }</td>
<td>PROCESS_WAIT_EVENT(); if (ev== PROCESS_EVENT_TIMER) { \3; }</td>
</tr>
<tr>
<td>task_signal(NET_RECEIVED, <strong>,</strong>*)</td>
<td>event void \2.receive(message_t *msg, error_t err) { \3; }</td>
<td>PROCESS_WAIT_EVENT(); if (ev== PROCESS_EVENT_COM) {</td>
</tr>
</tbody>
</table>
8.3 OS translator component

| | {
| task_signal(NET_SENTED,*, event void process_WAIT_EVENT();
| *) \2.sendDone(message_t*msg,
| \4.error_t err) {
| \3;
| } |

Table 8.3 Regular expressions and patterns for translating functions

8.3.3 Events translation

In order to create an event two actions are required. Firstly, the developer must use the task_signal primitive to declare the association between the event’s signal and the handler. The prototype of the task_signal is: task_signal(event, device, handler), where event is the event to manage, device is the associated device and handler is the name of the handler function. Then the developer must write the code for the handler. The mapping between events in the input code and the native code is done in different ways depending on the OS execution model.

In TinyOS the association between event handler and hardware interruption is defined explicitly, and the execution model is split-phase. To start, an application requires service from hardware. When data is available, the hardware will create an interruption as a response to the application. Next, the OS will stop any task in execution and call the corresponding event handler. When the handler has finished, other tasks are resumed. When translating the task_signal() to native code in TinyOS, the first parameter can be mapped to unique TinyOS event, the second parameter is mapped to an interface associated with the device. The code of the function identified in the third parameter will be translated and included as the handler for the event. For example, consider the following code line:

```c
task_signal(IO_READDONE, TEMPSENSOR, readDone)
```

The IO_READDONE represents the event occurred when a sensor has completed reading and the data has been made available. The TEMPSENSOR identifies the hardware that produces the interruption and readDone is the event handler. This code will be translated to the TinyOS 2.x native code snippet below:

```c
event void TempSensor.readDone(error_t err, val_t value) {
    //Event handler code
}
```
In Contiki, the blocking mechanism is used. A blocking is done through the macro `PROCESS_WAIT_EVENT`, which allows a protothread to stop and lets other protothreads execute. In general, when a protothread waits for an event, it is blocked. The process returns control to the OS kernel and waits for it to create an event signal. During this time the OS kernel is able to serve other processes. When the event occurs, OS kernel will send the event signal to the process, which then becomes unblocked. Then the code for the event handler will be executed. Therefore, the previously mentioned code line will be translated to the following Contiki native code snippet:

```c
PROCESS_WAIT_EVENT();

if (ev == event_data_ready) {
    // Event handler code
}
```
WSN simulator and measurement of non-functional properties

In our framework, in order to assess the quality of a candidate application we use simulative analysis method. For a given candidate solution which is specified by a set of adjustable parameter values, firstly we execute the solution on a simulator with a predefined experiment scenario. At the end of the experiment we measure the solution’s non-functional properties and the quality of the given candidate is specified on the basis of the result. In our work, only the measurable non-functional properties (refer to Section 7.2) are considered. This chapter discusses the selected WSN simulator and the measurement of these properties.

9.1 Cooja simulator

We choose Cooja [146] as the simulator because its architecture enables simultaneous simulation at different levels: network level, operating system level and machine code instruction level. At the lowest level, the instruction level, the complete node hardware platform is emulated using a discrete event microcontroller simulator. At the operating system level, a node is simulated by executing the native operating system code with its own data memory and operating system core library. At the network level, Cooja supports simulating a network of sensor nodes with different radio models and sleep duty cycles.

Cooja proposes three different radio models including: Unit Disk Graph Medium (UDGM) – Constant Loss, Unit Disk Graph Medium (UDMG) – Distance Loss and Directed Graph Radio Medium (DGRM).

- **UDGM – Constant Loss** is a radio model where the transmission range of a node is modeled as an ideal disk. All nodes within the disk can receive packets sent from this node, while the motes outside it do not receive the messages. The radius of the disk is defined by the product of the predefined maximal transmission range and the ratio of the current output power to the maximal output power of the simulated device. For example, if the maximal transmission range of a node is 100m and the current output power is half of the maximum then the disk has a radius of 50m.

- **UDGM – Distance Loss** is similar to the above one except that in this model the interferences and the success rate of sending/receiving data are considered. Firstly, if the interference distance is higher than the transmission distance, the interfered packets will be lost. Secondly, a packet is sent or received on the basis
of two probabilities, SUCCESS_RATIOTX (if this fails, no device receives the packet) and SUCCESS_RATIO_RX (if this fails, only the receiver does not receive the packet).

- **DGRM** is a radio model that allows users to specify the topology of the network through connection link between nodes. This model can be used to set the transmission success ratio and the propagation delay on the asymmetric links.

The architecture of COOJA is flexible and allows users to replace or extend many components with additional functionality. Example components are the radio media previously described, interfaces and plugins. The interface components are used to represent a node’s properties such as position, radio, battery or serial port. The plugin components provide the user with a GUI to interact with a simulation. They are Simulation Visualizer, Timeline, Log Listener and Contiki Text Editor.

- **Simulation Visualizer** is used to specify the network’s topology. User can add new nodes, change a node’s radio properties like transmission and interference range, and observe some interesting node information such as ID, type, attributes, log output, radio traffic, and LED states.

- **Timeline** is used to display one by one the state of the radio of each simulated node. Different colors are used to distinguish each radio state, for example, grey (on), blue (packet transmission), green (packet reception), and red (interfered).

- **Log Listener** plugin analyses every radio connection established between nodes during the simulation. The data reported by this plugin includes the sender and receiver identification, timestamp and the payload of every packet sent or received during the simulation.

- **Contiki Text Editor** is used to control the execution of the simulation scenario that is created using Java Script. User can use this plugin to pause the simulation, indicate the time out, automatically add/remove nodes, and interact with others plugins.

The simulation configuration is stored in a file using a clearly XML structure. User is allowed to save, load or change the file content during simulation. Figure 9.1 demonstrates an example of COOJA GUI.

Many interesting works have been done to extend COOJA with useful functionality. For example, an advanced version of the Timeline plugin [147] is developed to visualize the power consumption of the nodes in a simulated network. Other work presents the combination of YETI [27], an integrated development environment for TinyOS, and COOJA to support debugging WSN applications. However, COOJA still lacks the plugins to support collecting metrics about a connection between a pair of nodes such as the
number of frames lost/successfully transmitted from sender to receiver, the end-to-end delay when sending packets, the throughput in transmission/reception, the status of the queue in the relay nodes ... Without this information it is difficult to measure the application non-functional properties that are used to determine the quality of the candidate solution. Therefore, we develop COOJA plugin to collect some network metrics during the simulation and use this data to calculate the value of the non-functional properties.

9.2 WSN metrics collection and non-functional properties estimation

9.2.1 Power consumption estimation

Power consumption is the most important metric in WSN because it determines the lifetime of the network. In order to estimate the nodes energy consumption during a simulation, we employ the power profiling presented in [147] and [110] for Contiki OS and TinyOS respectively. In this method, a sensor node is recognized as a collection of components operating in distinguishable states. For example, the CPU is a component with two states (active mode or sleep mode), the radio transceiver is a component that can operate in transmission mode or listing mode, the sensor is another component that

![Figure 9.1 An example of COOJA GUI](image-url)
9.2 WSN metrics collection and non-functional properties estimation

has on/off operating mode. At a time $t$, the instantaneous power consumption of the
node can be estimated as the sum of the powers of all components operating in active
state. This value can be computed as:

$$P_{\text{node}}(t) = \sum_{i,j} P_{i,j} * S_{i,j}(t) \quad \text{Equation 9.1}$$

Where $P_{i,j}$ is the power consumption of the component $i$ operating in the state $j$, and
$S_{i,j}(t)$ will be 1 if the component $i$ is working in the state $j$, otherwise $S_{i,j}(t)$ will be 0.

The energy consumption can be estimated in the same way:

$$E_{\text{node}} = \sum_{i,j} P_{i,j} * T_{i,j} \quad \text{Equation 9.2}$$

Where $T_{i,j}$ is the time during which the component $i$ has operated in the state $j$.

The constant values $P_{i,j}$ are pre-defined for each hardware platform, for instance, Table 9.1 shows the values for the Zolertia Z1 mote. These values are specified based on the
hardware datasheet presented in [225].

<table>
<thead>
<tr>
<th>Component</th>
<th>Operating state</th>
<th>Power consumption (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Sleep mode</td>
<td>$1.5 \times 10^{-3}$</td>
</tr>
<tr>
<td></td>
<td>Active mode</td>
<td>15</td>
</tr>
<tr>
<td>Radio transceiver</td>
<td>Listening mode</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>Sending mode</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Receiving mode</td>
<td>60</td>
</tr>
<tr>
<td>Accelerometer sensor</td>
<td>Standby mode</td>
<td>$0.3 \times 10^{-3}$</td>
</tr>
<tr>
<td></td>
<td>Active mode</td>
<td>$4.5 \times 10^{-3}$</td>
</tr>
<tr>
<td>Flash memory</td>
<td>Deep Power Down mode</td>
<td>$3 \times 10^{-3}$</td>
</tr>
<tr>
<td></td>
<td>Active mode</td>
<td>30</td>
</tr>
<tr>
<td>Temperature Sensor</td>
<td>Standby mode</td>
<td>$3 \times 10^{-3}$</td>
</tr>
<tr>
<td></td>
<td>Active mode</td>
<td>$45 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

Table 9.1 The power consumption of Z1's components in different states

To capture the operating time of a component in a concrete state, some
instrumentation code must be added in the operating system level. We designed the
code for Contiki and TinyOS respectively to report the energy consumption to COOJA via
the Log Listener plugin. Since the energy measurement mechanism works in the
operating system level, so it does not affect the application running in the upper level.

In order to evaluate the accuracy of the energy consumption measurement method,
several experiments are conducted in [147] in which the author compares the reported
energy consumption with the oscilloscope energy measurements. The results show that
the power profiling method is accurate within 94% compared to hardware-based method.
9.2.2 Other network metrics collection

Beside the energy consumption, statistical data about the transmission between nodes like the frame loss rate, the throughput or delay ... must also be captured during the simulation for evaluation purposes. Therefore, we develop a plugin to collect and report the transmission data of the simulated network. Table 9.2 summarizes all collected metrics that are divided in two categories:

- Node metric group contains information about the transmission status of a single mote of the simulation.
- Link metric group includes the statistical data about the transmission between two motes in the network during the experiment.

<table>
<thead>
<tr>
<th>Metric group</th>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node metric</td>
<td>NM_FRAME_SEND_SUCCESS</td>
<td>Number of frames that a node sent successfully</td>
</tr>
<tr>
<td></td>
<td>NM_FRAME_SEND_NOT_SUCCESS</td>
<td>Number of frames that does not sent successfully by a node</td>
</tr>
<tr>
<td></td>
<td>NM_FRAME_RECEIVE_SUCCESS</td>
<td>Number of frames that is received successfully by a node</td>
</tr>
<tr>
<td></td>
<td>NM_FRAME_RECEIVE_NOT_SUCCESS</td>
<td>Number of frames that is not received successfully by a node</td>
</tr>
<tr>
<td></td>
<td>NM_PACKET_SEND_SUCCESS</td>
<td>Number of packets that a node sent successfully</td>
</tr>
<tr>
<td></td>
<td>NM_PACKET_SEND_NOT_SUCCESS</td>
<td>Number of packets that does not sent successfully by a node</td>
</tr>
<tr>
<td></td>
<td>NM_PACKET_RECEIVE_SUCCESS</td>
<td>Number of packets that is received successfully by a node</td>
</tr>
<tr>
<td></td>
<td>NM_PACKET_RECEIVE_NOT_SUCCESS</td>
<td>Number of packets that is not received successfully by a node</td>
</tr>
<tr>
<td></td>
<td>NM_BUFFER_STATUS</td>
<td>Number of frames remains in the sending queue</td>
</tr>
<tr>
<td>Link metric</td>
<td>LM_FRAME_SEND_SUCCESS</td>
<td>Number of frames that is delivered successfully from sender to receiver</td>
</tr>
<tr>
<td></td>
<td>LM_FRAME_SEND_NOT_SUCCESS</td>
<td>Number of frames that is not delivered successfully from sender to receiver</td>
</tr>
<tr>
<td></td>
<td>LM_PACKET_SEND_SUCCESS</td>
<td>Number of packets that is delivered successfully from sender to receiver</td>
</tr>
<tr>
<td></td>
<td>LM_PACKET_SEND_NOT_SUCCESS</td>
<td>Number of packets that is not delivered successfully from sender to receiver</td>
</tr>
<tr>
<td></td>
<td>LM_END_TO_END_DELAY</td>
<td>Packet delay when transmitting packets from sender to receiver</td>
</tr>
</tbody>
</table>

Table 9.2 List of WSN metrics collected by the plugin during a simulation
In Table 9.2, the term **FRAME** represents the actual radio packet sent on the radio medium using either **UDMG** or **DGRM**. The frame’s size is limited to 128 bytes in both Contiki and TinyOS. COOJA supports simulating the radio collisions and random errors on the channel when transmitting the frames; therefore, there is a frame loss rate when sending or receiving the frames. In order to capture the FRAME metrics, the plugin must listen to every new radio packet generated by the COOJA radio medium component. For each packet, the plugin will extract required information for the metrics and store them in an XML file when the simulation terminates.

Unlike **FRAME**, the **PACKET** represents the application packets that take place in a higher layer in the communication protocol stack. Packets are generated and exchanged by applications and have no size limit. If the size of a packet is bigger than 128 bytes, a fragmentation and reassembly mechanism must be used. Because COOJA does not implement any fragmentation and reassembly, the plugin has the function to generate original packets from the frames exchanged on the radio medium. Once a packet is generated its statistical data is extracted and stored as well.

The metric **NM_BUFFER_STATUS** is used to report the evolution of the sending queue (refer to Section 7.3.1) where motes keep the data to send. This value can be access using the **Buffer View** plugin in COOJA. The **LM_END_TO_END_DELAY** denotes the time taken for a packet to be transmitted from a sender to a receiver node in the simulated network. This metric value is obtained summing the transmission delay and the processing delay of each relay node in the path from sender to receiver. Suppose that there are K > 0 relay nodes (or hops) in the transmission route from sender to receiver. Then the end to end latency can be evaluated as:

\[
D = \sum_{i=1}^{K} (DT_i + DP_i)
\]

Where \(DT_i\) is the transmission time from hop \(i-1\) to hop \(i\), and \(DP_i\) is the processing time at the intermediate hop \(i\).

The value \(DP_i\) is archived by subtracting packet receiving timestamp from the sending timestamp at the relay node \(i\). The value \(DT_i\) is the latency for single-hop transmitting a packet. \(DT_i\) is depended on both the packet size that determines the number of fragmented frames and the time for single-hop transmission of each frame. Therefore the packet latency for single-hop transmitting can be evaluated as:

\[
DT = \sum_{j=1}^{N} DTF_j
\]

Where \(N\) is the number of fragmented frames and \(DTF\) is the time for transmitting a frame through single-hop radio medium. As mentioned in the previous paragraph, the plugin maintains all statistical data related to every packet; therefore the value \(N\) is
understood by the plugin. The value $DTF$ is varied based on the MAC protocol used in the simulation. In the following paragraphs we will analyze a common MAC protocol called X-MAC to formulate an estimation of $DTF$ value.

X-MAC [26], an asynchronous duty cycling MAC layer protocol, is the protocol used by sensor nodes in the simulation. It turns the radio on and off periodically during the wake mode and sleep mode. When node is in the wake mode, the radio turns on and waits for a sort preamble. When in the sleep mode, the radio switches off to save energy. Figure 9.2 illustrates the operation of X-MAC between two nodes, a sender and its receiver.

![Figure 9.2 The operation of X-MAC protocol](image)

In X-MAC, the sender will send a series of sort preambles that contains the receiver’s address before it starts transmitting the data frame. When a receiver wakes up, it listens to the radio medium to see if there is a transmitting preamble frame that matches its address. If there is no preamble or the receiver is not the target node, the receiver will switches off the radio and go back to the sleep mode again. Otherwise, the receiver remains awake and sends an Acknowledgement frame (ACK) to the sender to stop it sending the sort preambles. After the sender receives the ACK frame, it starts sending the data frame. Another ACK frame is used to confirm that the receiver has successfully received this data frame from the sender.

We simulated X-MAC protocol using COOJA. The Timeline of the simulation is showed in Figure 9.3, where Node 1 is the sender and Node 2 is the receiver. The gray bar in the figure means the radio is turned on, two nodes can only send or receive frames during this time. As seen in the timeline, nodes switch their radio on and off periodically to save energy. The blue bar is the process of sending the preamble, data or ACK frames. The green bar represents the process of receiving frames. When Node 1 has data to send, firstly it continues sending the sort preambles. When Node 2 wakes up and receives the preamble, it sends an ACK to Node 1. After Node 1 receives the ACK frame, it starts
sending the data frame. From Figure 9.2 and Figure 9.3 we can see that the simulation result matches very well with the operation of X-MAC in theory.

Figure 9.3 The Timeline of X-MAC simulation in COOJA

Based on the analysis above, we can estimate the time $DTF$ as the duration between two points in the sender timeline. The first point was the time when the sender started sending the first preamble frame, and the second point was the time when the sender received the last ACK frame to confirm that the receiver has successfully received the data frame. Since the plugin keeps track of all incoming and outgoing frame in each node during the simulation, it can estimate the $DTF$ value for each fragmented frame. By substituting the $DTF$ values back to equation 9.3, the plugin will figure out the latency for one-hop sending a packet, or the value of $DT$. In the next step, using equation 9.2 the value of $LM\_END\_TO\_END\_DELAY$ metric, or the delay when transmitting a packet from a sender to a receiver, is calculated by the plugin.

**9.2.3 Lifetime estimation**

Network *lifetime* is an important metric for the evaluation of WSNs. There are many different definitions and metrics that can be used to specify the lifetime of sensor networks [50]. In our research we used the definition that reflects the most critical situation in which all nodes are of equal importance and the death of a single node will result in the failure of the whole network. In this case, the lifetime is defined as the time from the beginning of the experiment until the moment when the first node runs out of battery. Thus, the network *lifetime* can be evaluated as:

$$T_{Life} = \min_{s \in S} \{T_s\} \quad \text{Equation 9.5}$$

where $T_s$ is the lifetime of a single node $s$ in the node set $S$. $T_s$ is the time interval from when the node is turned on until the moment when it runs out of battery.
9.2.4 Energy per delivered packet estimation

The average amount of energy spent per delivered packet points out the efficiency of the total consumed energy for transmitting data of the network because if a packet does not reach its destination, the energy spent in the transmission attempt is wasted. When the simulation is terminated, the plugin maintains the statistical data about the total energy consumption by each node and the total number of successfully delivered packets. The energy per delivered packet can be evaluated as:

\[ E_{pdp} = \frac{\sum_{s \in S} E_s}{P_{delivered}} \]  
Equation 9.6

where \( E_s \) is the energy consumption of a single node \( s \) in the node set \( S \). \( P_{delivered} \) is the total number of successfully delivered packets during the simulation.

9.2.5 Latency per delivered packet estimation

As discussed in the previous section, for each packet the metric LM_END_TO_END_DELAY denotes the delay when sending the packet from its source to its destination. At the end of the simulation, the plugin processes the individual delay for transmitting each delivered packet and calculates the average latency per delivered packets using the following formula:

\[ D_{pdp} = \frac{\sum_{p=1}^{P_{delivered}} D_p}{P_{delivered}} \]  
Equation 9.7

where \( D_p \) is the delay for successfully transmitting a single packet from its source to its destination, and \( P_{delivered} \) is the total number of successfully delivered packets during the simulation.

9.2.6 Packet loss ratio estimation

The packet loss ratio indicates the percentage of data packets that do not reach the packet destinations. We use the following formula to evaluate this ratio after the simulation:

\[ PLR = 1 - \frac{P_{delivered}}{P_{sent}} \]  
Equation 9.8

where \( P_{sent} \) is the total number of sent packets, and \( P_{delivered} \) is the total number of successfully delivered packets during the simulation.
Chapter 10  Optimization components

10  Optimization components

As mentioned above, one critical WSN design challenge involves meeting the non-functional constraints such as lifetime, reliability, throughput, delay, etc. In addition, WSN applications tend to have competing requirements, which exacerbates the design challenges. This chapter proposes approaches to optimize these non-functional properties in the context of WSN. In our approaches, the optimization problem means finding the best set of application adjustable parameters which satisfy an objective function of the non-functional constraints, from the search space of all feasible parameter values. In the first subsection, we formulate the optimization problem. The problem analysis is presented in the next subsection. In the last subsections, two solutions for the optimization problem and the design of the component are discussed.

10.1  Optimization problem formulation

10.1.1  Problem description

In order to optimize WSN application regarding to the non-functional properties, firstly developers must define a set of adjustable parameters whose values can be tuned to meet application requirements. These parameters and their possible value ranges are specified when creating the PIM of the application. In this step, the developers must also define an objective function to express the non-functional constraints such as maximization or minimization of a set of non-functional metrics. In the optimization process, each candidate application specified by a concrete set of adjustable parameter is executed on the simulator with a predefined simulation configuration. During the simulation, the developed simulator plugin will collect all experiment data and measure metrics like network Lifetime, Energy per delivered packet, Latency per delivered packet, or Packet loss ratio. The quality of the candidate application will be evaluated based on the metric values. In this way, the optimization problem means to find the best set of application adjustable parameters, which satisfy the objective function, from the search space of all feasible parameter values.

Because of the complex interrelationships between parameters, the large number of their possible values and the measurable responses, the optimization problem is not simple or idealized; actually it is a real-world problem with multiple inputs, multiple outputs, and multiple objectives. This multiple objective optimization problem will be formulated in the next subsection.
10.1 Optimization problem formulation

10.1.2 Problem formulation

We assume that the target application has a set of $N$ adjustable parameters $\{X_i\}_{i=1..N}$, and a set of $M$ non-functional metrics $\{P_j\}_{j=1..M}$. Each $X_i$ is an integer number that has a default value $X_i_{\text{Default value}}$ and a value range of $X_i_{\text{Min value}}$ - $X_i_{\text{Max value}}$. As previously mentioned the Default value, Min value and Max value of $X_i$ are predefined by the developer. The possible parameters values form a solution space, in which each candidate solution is a concrete set of adjustable parameter values $S_c=\{X_{C1}, X_{C2}, \ldots, X_{CN}\}$. The candidate solution is mapped to a point, specified by a set of evaluation metrics $P_c=\{P_{C1}, P_{C2}, \ldots, P_{CM}\}$, in the objective space. The mapping from the solution space $\{S\}$ to the objective space $\{P\}$ is defined experimentally by the simulation. Then the multi-objectives optimization can be formulated as the following:

$$\min \quad P(X) = [P_1(X), P_2(X), \ldots, P_M(X)]$$
subject to
$$X = [X_1, X_2, \ldots, X_N]$$
$$X_{i_{\text{Min value}}} \leq X_i \leq X_{i_{\text{Max value}}}$$

Equation 10.1

The multi-objective problem presented in Equation 10.1 can be solved by using a common technique named "linear scalarization"[125] that combines the multiple objectives into one single-objective function. In more detail, this technique minimizes a weighted sum of the objectives as below:

$$\min \quad P(X) = \sum_{i=1}^{M} w_i P_i(X)$$
subject to
$$X = [X_1, X_2, \ldots, X_N]$$
$$X_{i_{\text{Min value}}} \leq X_i \leq X_{i_{\text{Max value}}}$$
$$w_i \geq 0, \ i = 1 \ldots n$$

Equation 10.2

Weighting coefficients are chosen and maintained by the decision maker who is solving the optimization problem. In general, there is no a priori correspondence between the weighting coefficients and the solution vector. Therefore, it is difficult for the decision maker to be aware of which weights are the most appropriate to produce a good solution. In practice, the decision maker must try with different weight vectors in order to find out the ones that can produce a satisfactory solution. This is a technical shortcoming of the scalarization method because performing many optimizations with different weight vectors will significantly increase computational burden.

Another approach to solve the multi-objective optimization problem is to use the concept of Pareto optimality which is defined as follows:

- An objective vector $P_U=\{P_{U1}, P_{U2}, \ldots, P_{UM}\}$ dominates an objective vector $P_V=\{P_{V1}, P_{V2}, \ldots, P_{VM}\}$ if $P_U$ is better than $P_V$ with respect to at least one objective and not
worse than $P_V$ with respect to all other objectives, or represented in mathematic terms:

$$P_U \text{ dominates } P_V \iff \exists i \in \{1 \ldots M\} | P_{U_i} < P_{V_i} \land \forall i \in \{1 \ldots M\} | P_{U_i} \leq P_{V_i}$$

Equation 10.3

- A solution $s^*$ is said to be a Pareto optimum for the problem if and only if there is no $s \in S$ such that $P_i(s) \leq P_i(s^*)$ for all $i \in \{1 \ldots M\}$. Based on Equation 10.3 we can say a solution $s^*$ is a Pareto optimum if there is no solution $s_2$ such that the objective vector $P(s_2)$ dominates the objective vector $P(s^*)$.

A very common situation in the multi-objectives optimization problem is that an improvement in one objective will lead to degradation in one or more of the remaining objectives. In this case, the ideal optimal solution does not exist; instead, the answer for the problem is the set of all Pareto optimum solutions. This set of non-dominated solutions is named the Pareto frontier of the solution space.

Figure 10.1 An example of Pareto frontier

Figure 10.1 shows an example of the Pareto frontier in a two objectives optimization problem. In the example, A and B are two non-dominated solutions on the Pareto frontier. Solution A has a smaller value of Objective 1 than solution B, but a larger value of Objective 2. Conversely, solution B has a smaller value of Objective 2 than solution A, but a larger value of Object 1. Neither A nor B is better than the other. However, we can say the solution A dominates (or is better than) solution C. The same conclusion can be drawn when comparing two solutions B and D.
With the concept of Pareto optimality, solving a multi-objectives optimization problem is understood as computing or approximating the Pareto frontier and choosing the most appropriate Pareto optimum solution for the problem. In the next subsection, the analysis of the problem and two different methods for approximating the Pareto frontier are introduced.

### 10.2 Optimization problem analysis

#### 10.2.1 Analysis of the effects of adjustable application parameters

In this section we analyze relationships that exist between the adjustable parameters and the non-functional properties of WSN applications. We set up different experiment scenarios in which only a single adjustable parameter was changed. After each simulation we collected data on three non-functional property metrics: the *energy consumption per delivered packet*, the *latency per delivered packet* and the *packet loss ratio*. Based on the collected data, graphs were plotted of the evaluated parameter against the observed metrics to support distinguishing the relationships.

In order to ensure that the collected data from simulation experiments are valid and meaningful, each simulation is executed several times, during a reasonably long period and in a sufficiently large network. In more detail, a simulated network with around 200 randomly distributed nodes is used for testing the application. Each simulation is configured to run in three rounds and the simulation length for each round is 300 seconds. After each simulation, the average values of all observed metrics are computed and stored. Because the metrics are measured in different units or scales, we normalize all values such that for each metric the measured value range $[\text{min}, \text{max}]$ is mapped onto the range $[0,1]$.

The first experiment scenario is conducted to investigate a popular data broadcasting protocol in WSN named gossip routing [77]. Gossip protocol is frequently used when a source node has to send some data to all nodes or a set of particular nodes in the network. This is achieved by applying a flooding scheme in which each node broadcasts data packets to other nodes within its transmission range. Flooding in wireless networks benefits from the nature broadcast property of the transmission medium. This property enables nodes to transmit only one packet to all neighbors, instead of one packet per dedicated link, as it is the case in wired networks. The main drawback to the flooding scheme is that it often produces many redundant packets when broadcasting. Gossip introduced a probabilistic rebroadcast scheme to address the problem with flooding. In a simple gossip protocol, after receiving a packet, a node continues broadcasting this packet with a given probability $p$ or drops the packet with probability $1-p$. 
Clearly, choosing a correct value of $p$ is an important step when designing a gossip-based routing protocol. In order to evaluate the impacts of the rebroadcast probability $p$ on the non-functional metrics of the application, 10 experiments are conducted with 10 different values of $p$ distributed evenly in the interval $[0, 1]$. In each experiment, we configured a simulated network of 200 nodes which are distributed randomly in an area of 500 x 500 meters. Figure 10.2 (a) shows the first experiment scenario, where each node has a transmission range and interference range represented as a green circle and gray circle respectively. When a node transmits packets, only nodes in the green area are able to receive the packets. Nodes in the gray area could not receive the packets and they are also interfered which means that they are not able to receive and transmit simultaneously. The simulated network has a source node that frequently sends packet to its neighbors. Other nodes are implemented with the simple gossip routing protocol to continue retransmitting the packets. The radius of the transmission range and the interference range of each node in the simulation are 50 meters and 80 meters respectively.

Sensor nodes often reserve their power by switching between sleeping and working states. When a node is in the sleeping state, its radio is turned off and node is not able to receive or send packets. Otherwise, when a node is in the active state it is fully functional. We conduct the second experiment scenario to evaluate the impact of the sleep time on the non-functional metrics of the application. The simulated sensor nodes in this experiment are similar to the nodes used in the first scenario. In this case, each node serves as a source node and a relay node to send packets to a sink node. In more detail, every node except the sink will periodically send packets to the sink. A data
collection tree routing protocol [73] is implemented in each node to support forwarding the packets. The simulated nodes are configured to switch between sleep mode and active mode with a predefined sleep time. Twelve experiments are conducted with twelve different values of the sleep time (in second) distributed evenly in the interval [0, 60]. Figure 10.2 (b) demonstrates the experiment configuration in the second scenario.

For each scenario we have the objective of minimizing the desired non-functional properties measured by the corresponding metrics such as energy consumption per delivered packet, packet loss ratio or latency per delivered packet. However, the experiment data shows that no single value of the adjustable parameter satisfies this objective. Instead, in order to obtain an acceptable compromise solution, a multi-objective optimization technique is required. The experiment results are discussed in the next sections.

10.2.2 The impacts of rebroadcast probability on the application non-functional properties

Figure 10.3 shows the measurement values of the three non-functional metrics mentioned above corresponding to various value of the rebroadcast probability $p$. Figure 10.3 (a), (b) shows that in general the packet loss rate decreases when increasing $p$, it means the network will become more reliable with bigger $p$ values. Furthermore, from the simulation result we observe the bimodal behavior of a gossip-based network in which either most nodes receive a packet, or only a few do.

The bimodal behavior is well studied in [77], the author has pointed out that there is a sharp transition in the packet loss rate values at a critical value of $p$. Our experiment result agrees to this conclusion and indicates that the critical value is somewhere from 0.4 to 0.5. When $p$ is bigger than 0.7, the packet loss rate is steady at around 15%. We notice that the average packet loss rate could not reach 0, even though $p$ is 1, because of the interference. As network activity increases with $p$, the interference also occurs more frequently and this causes packet drop for receiving nodes within the interference range.

In contrast, the average energy spent on transmitting a delivered packet increases when increasing $p$. It can be explained as: when $p$ value is high, many redundant packets are produced in the network. After a packet has been received, the energy spent on retransmitting it again is wasted; therefore the average energy cost for successfully transmitting a packet will increase when increasing the rebroadcasting probability $p$. When $p$ is 0, every node is not able to receive packets except the ones that are one-hop away from the source node. In this case the total consumed energy is small, but the
number of delivered packet is also small as well. When $p$ increases, the energy consumption per delivered packet rises slowly and the network becomes more power inefficient as illustrated in Figure 10.3 (a), (b).

![Figure 10.3](image_url)

**Figure 10.3** The impacts of rebroadcast probability on applications non-functional properties

The changes of $p$ values also affect the average end-to-end latency of a gossip-based network, as shown in Figure 10.3 (b), (c). As $p$ increases from 0 to 1, the end-to-end delay also increases slowly due to the interference between nodes and their limited queue size. The analysis above shows that we can improve the network reliability by increasing the rebroadcast probability; however it will introduce more end-to-end delay and power consumption. In additional, the magnitude of change in the energy and the latency values is fairly small, around 40 percent as illustrated in Figure 10.3 (b). It means that rebroadcast probability $p$ may have greater impacts on the network reliability than the delay or energy consumption.
10.2 Optimization problem analysis

10.2.3 The impacts of sleep time on the application non-functional properties

Figure 10.4 illustrates how different sleep time durations affect the reliability, energy consumption and latency metrics of WSN applications. As shown in Figure 10.4 (a) and (b), a duty-cycling mechanism which consists in putting sensor nodes in sleep mode during idle periods is an effective approach to energy conservation. This result agrees to the findings in [167], in which the author states that in a sensor node the radio subsystem typically consumes much more power than the sensing and processing component. As seen in the figure, with a small range of sleep time values (smaller than 15 seconds), the average energy spent on each delivered packet decreases dramatically when increasing the sleep time. However, with a bigger value range of the sleep period, increasing the sleep time would not allow node to save more energy consumption. One possible reason of this finding can be related to the increment of the overhead cost in packet transmission and routing when nodes stay longer in the sleep mode. In addition, increasing the sleep time introduces more latency and packet loss in the network. In fact, as illustrated in Figure 10.4 (c), the average latency per delivered packet and the packet loss rate increase in almost direct proportion to the sleep duration.
Based on the analysis presented in the above sections we might conclude that the adjustable parameters have complex influences on the non-functional metrics of the WSN application. Moreover, the impacts strongly depend on the application context as well. Therefore, in order to obtain an acceptable compromise solution to the problem of minimizing all the desired non-functional properties, a multi-objective optimization technique is required.

10.3 Optimization solution approaches

10.3.1 Systematic Statistical Design of Experiments (SSDOE) approach

In order to deal with the optimization problem presented in Section 10.1.2, two different approaches are applied in this thesis. The first approach is Systematic Statistical Design of Experiments (SSDOE) [22, 150] which is a well-known experiment design theory. SSDOE is applied to gain a better understanding of the potential relationship between input variables and output responses. This approach requires the identification of factors (inputs to an experiment with different values or levels) and responses (outputs of the experiment, observations or measures). A series of experiments is run with all possible combinations of all acceptable values of the factors and the responses are recorded. Then statistical methods are used to create a model to predict the response values. The SSDOE could be considered as a broad but shallow search method which explores the search space in a systematic and even manner. The main disadvantage of SSDOE method is that the experiment cost will grow rapidly when there are many factors and each factor has many acceptable values. To address this issue, we apply the multi-step strategy as the follows:

- **Step 1**: For each adjustable parameter $X_i \ (i=1 \ ... \ N)$, we select a small set of representative values $R_{iK} = \{X_{i1}, X_{i2}, ..., X_{iK}\}$. The sample set of the solution space is formulated as the combination of all selected representative values, $S_R = \{X_{R1}, X_{R2}, ... X_{RN}\}$, where $X_{Ri}$ is taken from the set of representative values $\{R_{iK}\}$ for that adjustable parameter $X_i$. For each point $S_C$ in the sample set $\{S_R\}$ we run the simulation to measure the corresponding objective values $P_{C} = \{P_{C1}, P_{C2}, ... P_{CM}\}$. The set of all measured objective values $\{P_R\}$ is the sample set of the objective space. We now have the mapping $\{S_R\} \rightarrow \{P_R\}$, an approximation of the mapping $\{S\} \rightarrow \{P\}$ in a low resolution, where $\{S\}$ is the solution space and $\{P\}$ is the objective space.

- **Step 2**: With the finite set of sample points in the step 1, we estimate the relationship between adjustable parameters and non-functional metrics by fitting a statistical model to the known sample points to approximate the relationship. For each non-functional metric $P_m$, a separate linear regression
model is created. We can then use these models to predict the network responses for any arbitrary set of input parameter values.

- **Step 3**: We increase the number of representative values by sampling each parameter at high resolution. The solution space now contains every valid combination of every sampled value of all adjustable parameters. Each combination represents a single candidate solution in the solution space. The statistical model developed in Step 2 is used to predict different non-functional metric values corresponding to every candidate solution. The metric values will form the objective space of the problem.

- **Step 4**: We examine all points in the objective space to identify the Pareto frontier that is the result of the optimization problem. We run the simulation again with the input set that corresponds to each point in the Pareto frontier. The simulation results are used to validate the Pareto frontier’s points to avoid archiving meaningless and unusable outcome after the optimization stage.

The main computational burden of the optimization process is the evaluation of the candidate solutions one by one through simulation. To reduce the computation cost, in the first step, we only choose a small set of representative values for each input adjustable parameters. The chosen values of a parameter must include at least the Default value, Minvalue and Maxvalue. Other representative values are sampled evenly over the range from Minvalue to Maxvalue. We then compose a test suite that contains every valid combination of all selected representative values for all input parameters. This test suite \( \{SR\} \) is a subset of the solution space \( \{S\} \), where each solution candidate is a valid combination of possible parameter values. We executed a simulation experiment for each input point in the subset \( \{SR\} \) to collect the set of all measured non-functional metrics \( \{PR\} \) which is a subset of the objective space \( \{P\} \). The mapping \( \{SR\} \rightarrow \{PR\} \) represents partially the relationship between the input solution space \( \{S\} \) and the output objective space \( \{P\} \). Other parts of the mapping can be inferred by fitting a statistical model to the known points in \( \{SR\} \) and \( \{PR\} \), as mentioned in Step 2.

We use linear regression models to investigate the mapping \( \{S\} \rightarrow \{P\} \). For each non-functional metric \( P_m \) a model is created to approximate the relationship between the metric and all input adjustable parameters. The following equation is used to develop the fitted models:

\[
P_m = \beta_0 + \sum_{i=1}^{N} \beta_i X_i + \varepsilon \quad \text{Equation 10.4}
\]

where \( \beta_0 \) is the intercept (constant), \( \beta_i \) is the coefficient for parameter \( X_i \) and \( \varepsilon \) is the error term. The value of \( \beta_0, \beta_i, \varepsilon \), can be estimated using the known points in \( \{SR\} \) and \( \{PR\} \).
With the statistical model developed in Step 2, we are able to increase the number of representative points in the solution space. In Step 3, we sample the search range of each input adjustable parameter in high resolution to select a greater number of representative points. As the same as the first step, all representative values are sampled evenly over the search range, and the test suite \( \{S_R\} \) is composed of every valid combination of all representative values. For each solution candidate in \( \{S_R\} \), we interpolate the corresponding point in the objective space \( \{P_R\} \) by substituting \( X_1, \ldots, X_N \) values into the fitted model for each non-functional metric \( P_m \) represented in Equation 10.4. Thus, we now have \( \{P_R\} \), a high resolution approximation of the objective space \( \{P\} \).

In Step 4, we examine all points in the representative objective space \( \{P_R\} \) to indicate the Pareto frontier that is the result of the optimization problem. We develop two algorithms: the first one is used to check whether a point \( u \) is dominated by a point \( v \) in the space \( \{P_R\} \); and the second one is used to find out all non-dominated points in the Pareto frontier from the available points in the set \( \{P_R\} \). Two algorithms are presented in Listing 10.1 and Listing 10.2 respectively.

**Listing 10.1 Algorithm checks whether a vector \( u \) is dominated by a vector \( v \)**

**Input:** \( u \) and \( v \) are two points in the objective space  
**Output:** True if \( u \) is dominated \( v \), otherwise the algorithm returns False

```
for i=1 \rightarrow M do
    if v[i] > u[i] then
        return False
    endif
return True
endfor
```

**Listing 10.2 Algorithm finds all points in the Pareto Frontier**

**Input:** The set \( \{P\} \) contains many points in the objective space  
**Output:** The set \( \{PF\} \) includes all points in the Pareto Frontier  
**Algorithm:**

\( PF = \emptyset \)

```
for all \( u \) in \( P \) do
    isNonDominated = True
    for all \( v \) in \( P\setminus\{u\} \) do
        if isDominated(\( u, v \)) then
            isNonDominated = False
            Stop the second for loop
        endif
    Endif
```
10.3 Optimization solution approaches

Finally, in order to verify the optimization results, we run the simulation again with the input set that corresponds to each point in the Pareto frontier. Then the simulation results are used to validate Pareto frontier points to avoid archiving meaningless and unusable outcome after the optimization stage.

10.3.2 Multi-Objectives Evolution Algorithm (MOEA) approach

Alternative approaches, which are better suited to large numbers of input parameters, are also considered in this thesis. For example, an evolutionary algorithm (EA), inspired by Darwin’s principle of natural evolution, could be a feasible candidate. Starting with a population (a set of initial candidate solutions), in each generation (algorithm’s iteration), the parents (best solutions) are chosen and used to generate offspring (new solutions) by crossover (recombining the information of two parents in a new way) or mutation (randomly modifying a parent). The new generated offspring are then included into the population, replacing some of the weaker individuals (solutions in the set). By repeatedly choosing the better solutions and using them to create new candidates, the population is evolved, and the solution quality is increased gradually, just like in nature, where the individuals become better and better adapted to their environment through evolution. The advantage of EAs is that they can work very well with many complex objective functions and constraints, making very few assumptions, and even without a mathematical description of the problems. EAs have proven to be very effective, and are successfully applied on a wide variety of complex optimization problems such as logistic, scheduling or engineering design.

EAs are extended to Multi-Objective Evolutionary Algorithms (MOEAs) to deal with multi-objective problems. The differences between MOEA and single objective EA are how they rank individuals and choose parents from the population. In a single objective EA the individuals are ranked based on this objective, and the best individuals are selected as the parents. In MOEA, there is no obvious way to rank the individuals because of multiple objectives, and many different ranking and selection schemes have been developed to produce a good approximation to the Pareto frontier. In a MOEA the solutions may be arranged according to the distance of the solution to the true Pareto frontier, the range or the distribution of the solution. Compared to other heuristic algorithms, MOEA are able to search for a set of solutions as a result. In the context of
Chapter 10  Optimization components

multiple objectives it means that MOEA can search for a representative set of Pareto-
optimal solutions, approximating the true Pareto frontier in a single run. After the
optimization users are able to pick a solution of their interest from the Pareto set, taking
into account the possible tradeoffs between competing objectives.

We apply a MOEA to solve the optimization problem. An optimization component based
on SPEA2 [224] is integrated in our framework to support developers in optimizing their
applications in terms of non-functional constraints. SPEA2 is an improved version of the
Strength Pareto Evolutionary Algorithm (SPEA) which uses a ranking procedure to assign
better fitness values to non-dominated solutions that encourage uniform distribution of
the population near the Pareto frontier. SPEA uses a regular population \( P \) and an
external set called an archive set \( E \). The archive set stores all non-dominated solutions of
the search space that have been investigated so far during the search. In each iteration,
fitness values are assigned to both archive and population members:

- Each individual \( y \) in the archive \( E \) is assigned a strength value defined as:

\[
s(y, t) = \frac{np(y, t)}{N_p + 1}
\]

Equation 10.5

where \( np(y, t) \) is the number of individuals that \( y \) dominates in the population \( P \)
and \( N_p \) is the size of the population \( P \).

- The fitness of the individual \( y \) is assigned as the strength value of \( y \): \( f(y, t) = s(y, t) \).

- The fitness of the individual \( x \) in the population \( P \) is calculated as:

\[
f(x, t) = 1 + \sum_{\substack{y \in E \\ y \text{ dominates } x}} s(y, t)
\]

Equation 10.6

In SPEA2, a fine-grained fitness assignment strategy which incorporates an estimation of
density of the Pareto frontier is employed. In detail, the strength value is assigned to
individuals in both the archive set \( E \) and the population \( P \). Then the fitness of the
individual \( x \) is calculated as:

\[
f(x, t) = \sum_{\substack{y \in E \cup P \\ y \text{ dominates } x}} s(y, t) + D(x, t)
\]

Equation 10.7
where \( \mathbf{D}(\mathbf{x}, t) \) is the density estimation of the individual \( \mathbf{x} \). \( \mathbf{D}(\mathbf{x}, t) \) is evaluated using an adaptation of the k-th nearest neighbor method [224].

\[
D(\mathbf{x}) = \frac{1}{\sigma^k_x + 2}
\]

Equation 10.8

where \( \sigma^k_x \) is the Euclidean distance of the objective values between the solution \( \mathbf{x} \) and its k-th nearest neighbor, and \( k \) is chosen as the square root of the size of the solution set \( \mathbf{P} \) and archive set \( \mathbf{E} \) combined.

The complete pseudo-code of the SPEA2 algorithm is presented in the Listing 10.3 Strength Pareto Evolution Algorithm 2 below.

<table>
<thead>
<tr>
<th>SPEA2 pseudocode</th>
</tr>
</thead>
</table>
| **Input:** \( \mathbf{P}_N \) (population size)  
\[ \mathbf{E}_N \) (archive size)  
\[ T \) (maximum number of generation)  

**Output:** Set of non-dominated solutions  

\[ \mathbf{P} \leftarrow \text{InitializePopulation}(\mathbf{P}_N) \]

\[ \mathbf{E} \leftarrow \emptyset \]

**while** (\( \neg \)StopCondition(\( T \))) **do**

**for** \( \mathbf{S}_i \in \mathbf{P} \) **do**

\[ \mathbf{S}_i \text{Objective} \leftarrow \text{EvaluateObjective}(\mathbf{S}_i) \]

**endfor**

\[ \mathbf{U} \leftarrow \mathbf{P} + \mathbf{E} \]

**for** \( \mathbf{S}_j \in \mathbf{U} \) **do**

\[ \mathbf{S}_j \text{Fitness} \leftarrow \text{CalRawFitness}(\mathbf{S}_j, \mathbf{U}) + \text{CalDensity}(\mathbf{S}_j, \mathbf{U}) \]

**endfor**

\[ \mathbf{E} \leftarrow \text{GetNonDominated}(\mathbf{U}) \]

**if** \( \text{Size}(\mathbf{E}) < \mathbf{E}_N \) **then**

\[ \text{FillArchiveWithBestRemaining}(\mathbf{E}, \mathbf{E}_N, \mathbf{U}) \]

**elseif**

\[ \text{RemoveMostSimillar}(\mathbf{E}, \mathbf{E}_N) \]

**endif**

\[ \text{Parents} \leftarrow \text{SelectParents}(\mathbf{E}) \]

\[ \mathbf{P} \leftarrow \text{CrossoverAndMutation}(\text{Parents}) \]

**endwhile**

**return** \( \text{GetNonDominated}(\mathbf{E}) \)

Listing 10.3 Strength Pareto Evolution Algorithm 2

In the listing, the function \( \text{EvaluateObjective()} \) accesses the simulator interface to estimate the objective values (non-functional metrics) for each solution. \( \text{CalRawFitness()} \) and \( \text{CalDensity()} \) functions are used to evaluate the raw fitness of a given solution (the sum of strength of the solutions that dominate this solution) and the density of the
related area of the Pareto frontier. The fitness value of a solution, is then assigned as the sum of the raw fitness and the density value. The \textit{FillArchiveWithBestRemaining()} function is used to fill the archive set with remaining candidate solutions in order of fitness value. The function \textit{RemoveMostSimilar()} truncates the archive set by removing those members with the smallest value of $\sigma^k$ as previously calculated. The binary tournament selection method is implemented in the function \textit{SelectParents()} to choose parents from the archive set. The \textit{CrossoverAndMutation()} function is used to perform the crossover and mutation operators on the selected parent individuals.

One of the first decisions that have to be taken when using a MOEA is to define how to encode or represent the solutions of the problem to solve. In our problem, a solution candidate is composed of set of real numbers corresponding to the set of adjustable parameter values. Therefore, we used the real-coded method [83], where an individual (a solution in the solution space) is a vector of floating point numbers:

$$X = \{x_1, x_2, ..., x_N\}$$

for all $i = 1 \ldots n$, $x_i^L \leq x_i \leq x_i^U$

where: $x_i^L$ – lower boundary of the $i^{th}$ gene,

$x_i^U$ – upper boundary of the $i^{th}$ gene

Equation 10.9

In order to generate new solutions from existing ones, EAs use selection, crossover and mutation operators:

- The selection operator chooses individuals from the population for later breeding. We use binary tournament selection with replacement in which two individuals are taken at random and the better individual is selected from the two. After selecting the better ones, the two individuals are immediately replaced into the population for the next selection operation.

- The crossover operator combines the features of two parent individuals to form two offspring, with the possibility that good individuals may generate better ones. We implement Simulated Binary Crossover (SBX) operator [46] that is applied to a pair of parent individuals: $X^1 = \{x_1^1, x_2^1, ..., x_N^1\}$ and $X^2 = \{x_1^2, x_2^2, ..., x_N^2\}$. In this operator, the following steps are involved to generate two offspring solutions: $C^1 = \{c_1^1, c_2^1, ..., c_N^1\}$ and $C^2 = \{c_1^2, c_2^2, ..., c_N^2\}$.

\textbf{Step 1:} For each variable $x_i$ a number $u_i$ is created randomly between 0 and 1.

\textbf{Step 2:} A probability distributed function is specified to create offspring solutions so that they have the same search power as that in a single-point crossover in binary-coded method. Then find a number $\beta_i$ so that the area under the probability curve is equal to the number $u_i$. As figured out in [46], $\beta_i$ is calculated using the following formula, where $\eta$ is a parameter called the distribution index
for crossover. The larger value of $\eta$ the higher probability for generating a near-parent offspring, and vice versa. It is common in the literature to set $\eta$ to 20:

$$
\beta_i = \begin{cases} 
\frac{1}{(2u_i)^{\eta+1}}, & \text{if } u_i \leq 0.5 \\
\frac{1}{2(1-u_i)^{\eta+1}}, & \text{otherwise}
\end{cases}
$$

Equation 10.10

**Step 3**: Compute the offspring as:

$$
c_i^1 = 0.5[(1 + \beta_i)x_i^1 + (1 - \beta_i)x_i^2] \quad \text{and} \quad c_i^2 = 0.5[(1 - \beta_i)x_i^1 + (1 + \beta_i)x_i^2]
$$

Equation 10.11

- The mutation operator is used to maintain the diversity of the population in EA. It operates on a single selected individual at a time and modifies the individual independent of the rest of the population members. Among the many mutation operators for real-coded individual that are investigated in [168], we implement the polynomial mutation operator in our approach. In this operation the following steps are used to generate the offspring $C = \{c_1, c_2, ..., c_N\}$ from the parent individual $X = \{x_1, x_2, ..., x_N\}$:

**Step 1**: For each variable $x_i$ a number $u_i$ is created randomly between 0 and 1.

**Step 2**: Using a polynomial probability distribution function:

$$
P(\delta) = 0.5(\eta + 1)(1 + |\delta|)^\eta
$$

to calculate the parameter $\delta_i$ as:

$$
\delta_i = \begin{cases} 
\frac{1}{(2u_i)^{\eta+1}} - 1, & \text{if } u_i \leq 0.5 \\
\frac{1}{2(1-u_i)^{\eta+1}}, & \text{otherwise}
\end{cases}
$$

Equation 10.12

Where $\eta$ is called as distribution index for mutation parameter. It is common in the literature to set $\eta$ to 20.

**Step 3**: Calculate the offspring as:

$$
c_i = x_i + \delta_i(x_i^U - x_i^L),
$$

where $x_i^U$ and $x_i^L$ are the upper bound and lower bound of $x_i$

Equation 10.13
10.4 Optimization component design

The design of the optimization component is presented in Figure 10.5 below. Application developers can interact with the component via a Command-line User Interface. To start the optimization process, the developer must specify some configuration files such as a simulation scenario, an application configuration and corresponding PIM that describes the input parameters and the non-functional properties he wants to optimize. The developer is suggested to verify the PIM before the optimization stage to ensure that the application meets all its functional requirements. This verification can be done by testing the application with the simulation using a proper scenario.

During the optimization process various sets of adjustable parameter values are generated and stored into temporary files. The Application Translation Interface updates the PIM with the generated adjustable parameter values and executes the translation component and the compiler to create a new binary image of the application. The Simulation Interface is designed to interact with Cooja simulator. It is a multi-threading application that is able to launch a maximum of 8 simulations in parallel. Each simulation is run with an instance of Cooja simulator because Cooja is a single-thread program. The Simulation Interface takes as the input the application’s binary image and the simulation scenario, after that it generates an equivalent saved simulation file for Cooja and run the
simulation with a new instance of Cooja simulator. When the simulation is finished, the *Simulation Interface* updates the optimization component with the simulation result.

We implement two different components for two optimization approaches SSDOE and MOEA presented in section 10.3.1 and section 10.3.2 respectively. The *SSDOE component* is not completely automatic. Actually, the component only performs the first step mentioned in the SSDOE approach, in which a small set of representative values for each input parameter is created and evaluated in the simulator. Other steps must be done manually by the application developer using some statistical analysis tools like R, MATLAB or SPSS.

On the contrary, the *MOEA component* is a completely automatic process, where an initial solution set is evolved continuously until the termination condition is met. The MOEA component is developed based on JMetal [55] – a popular object-oriented Java framework for multi-objective optimization with meta-heuristics. We simply extend JMetal with our solution representation model, new crossover and mutation operators and a new fitness evaluation scheme. During the optimization, the simulation results are stored separately from the optimization results. This information can be used by the developers for the further analysis.
Part III
Evaluation and Conclusion
11 Framework evaluation

This chapter presents the evaluation of the proposed development framework. The goal of our evaluation are to assess the ease with which novice and intermediate users can develop correct and power efficient WSN applications, the portability level achieved by developing applications at a high-level of abstraction, and finally estimate the overhead due to the framework usage in term of the footprint and executable code size of the application.

We addressed the first goal by conducting a user study involving novice and intermediate programmers to evaluate how the target users respond to the framework, and to determine how quickly and correctly they complete programming tasks using the development model. To accomplish the other goals, on one hand, we selected different target platforms (operating system and hardware) and some typical WSN applications. On the other hand, we programmed the applications from scratch using our framework and by using each of the operating system individually. By comparing the difference between the development processes, we see that our MDD framework is able to provide a reproducible and portable application development environment to a WSN with heterogeneous nodes. Additionally, the overhead due to the framework usage is negligible, as shown in the experiment result. The details of our findings are discussed in the subsections in this chapter.

11.1 The ease of WSN programming

11.1.1 User study setup

To study the ease at which novice and intermediate programmers can develop correct WSN applications using our MDD framework, we conducted a user study with 50 participants recruited from recent graduate and undergraduate students at Hanoi University. The participants are classified as novice users and intermediate users. The novice users are first or second year students who have no or little programming experience. In contrary, the intermediate users have some programming experience and training. They are final year or recently graduated students who have participated in some programming courses about C/C++ and Java language. The participants are randomly assigned to one of three groups. The first group developed WSN applications using the MDD framework, while other groups used TinyOS and Contiki OS respectively. Figure 11.1(a) shows the novice and intermediate members in each group, and Figure 11.1(b) shows the size of the largest program in term of the total lines of code written by each intermediate user. As illustrated in the figures, participants with equivalent programming ability are chosen for each group.
We organized a six hours tutorial section for each group at the beginning of the study. The purpose of the tutorial is to help the participants familiarize themselves with the sensor hardware, the language and the programming model. The tutorial content is specifically designed for each group. For instance, we presented to the first group a tutorial about our MDD framework and how to use the IDE to define a PIM for a WSN application. For the second group the basis of TinyOS, its event-driven programming model and NesC language are introduced. The third group is given an introduction on how to develop an application for Contiki using C language. All groups have a common tutorial section about using Cooja simulator and how to simulate a Z1 mote with Cooja. For every group, during the tutorial, the users can practice writing example programs, then compiling and executing them on a test bed or on the simulator.

After the tutorial sections, the participants are given three exercises that must be completed within three hours. The exercises are designed such that no solution can be obtained by simply modifying the tutorial example code. Because we want to force the participants to apply the language primitives and the programming model learned from the tutorial to deal with the exercises. The exercises are presented below, ranked by increasing difficulty:

- **Exercise 1: Read Sensor** – this exercise asks the participant to write an application that reads data from temperature sensor at the rate 1 Hz and prints data (for debugging).
- **Exercise 2: Broadcasting** – the participant is asked to write an application that creates a message and sends it to mote’s neighbors.

![Figure 11.1 Groups' members and their programming background](image-url)
Exercise 3: *Multi-hop sending* – in this exercise, the participant must write a program that periodically creates a message at the rate 1 Hz and sends it to mote’s neighbors. Every time, a mote receives a message from its neighbors, it will continue forwarding the message to the network.

After time limit expires, we collected all participants’ solutions and evaluated their answers. For the sake of simplicity, we only considered whether the participants could fulfill the tasks with a correct solution without evaluating the quality of the solution in detail.

### 11.1.2 User study result

In Table 11.1 we present a summary of our user study result. Figure 11.2 illustrates the result in detail focusing on the exercises one by one.

<table>
<thead>
<tr>
<th>Exercise</th>
<th>MDD framework</th>
<th>TinyOS</th>
<th>Contiki OS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Novice user</td>
<td>Intermediate user</td>
<td>Novice user</td>
</tr>
<tr>
<td>Read sensor completed</td>
<td>7</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Broadcasting completed</td>
<td>4</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Multi-hop sending completed</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 11.1 Summary of the user study result*

As Figure 11.2(a) shows, for the first group, all intermediate programmers were able to complete the *Read Sensor* exercise, while the completion rates for novice programmers in the same group is 78%. In contrast, the completion rates for the second and the third group are 25% and 44% for novice programmers, as well as 50% and 75% for intermediate programmers, respectively. This trend continues for the participants attempting the second and the third exercise. As shown in Figure 11.2 (b) and (c), 44% of novice programmers using our framework were able to complete the *Broadcasting* exercise, whereas only 12% of novice programmers using TinyOS and Contiki could do so. Similarity, 75% of intermediate users in group 1 completed the second exercise, while the completion rates for intermediate users in group 2 and 3 are 25% and 50% respectively. For the most difficult exercise, *Multi-hop Sending*, 22% of novice programmers in the first group were able to complete the task, while none the of novice participants in the second group and 12% of novice participants in the third group succeeded. Specially, 63% of intermediate programmers in the first group were able to complete the most difficult task compared to 12% of the second and the third group. Overall, as shown in Figure 11.2(d), near half of exercises were done by the novice users,
who have no or little programming experience, using our framework. This completion rate is noticeably larger than the rate in TinyOS group (13%) and Contiki group (22%). Likewise, intermediate users in group 1 were able to complete 79% of all exercises, compared to 29% in group 2 and 46% in group 3, respectively.

![Figure 11.2 Detail of the user study result](image)

Finally, the finding from the user study can be summarized as the following:

- Novice and intermediate users are able to quickly get familiar with and use our framework to develop a typical WSN application that requires sensor data acquisition and network communication.
- While completion rate is task dependent, 78% of novice programmers were able to complete a simple task using our framework, whereas only 25 – 44% novice users are likely to do so using TinyOS and Contiki.
- With our framework, intermediate programmers were able to complete 79% of all exercises. In contrary, TinyOS and Contiki programmers finished only 29% and 46% of all assigned task.
• Developing a NesC program for TinyOS, coupled with its event-driven programming model, is the most cumbersome task for the participants in our study.

11.2 Portability evaluation

11.2.1 Mote platforms and benchmark applications

In order to determine the contribution of the MDD framework to WSN applications portability, firstly we selected a set of wide-range mote platforms and a set of typical WSN applications as a benchmark. Then, we developed a PIM for each benchmark application and used the framework to translate the models into platform-specific codes for all selected platforms. Clearly, our framework’s portability level can be evaluated by the percentage of WSN mote platforms for which the target code can be generated, and the percentage of WSN applications for which the framework is able to perform a correct translation to all selected platforms.

We selected nine hardware platforms and two operating systems to test our framework portability. Furthermore, there are two simulated mote types for TinyOS and Contiki OS respectively. The simulated motes have no specific hardware and exist only on Cooja simulator. The list of evaluated platforms is presented in Table 11.2.

<table>
<thead>
<tr>
<th>Hardware platform</th>
<th>Operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TinyOS</td>
</tr>
<tr>
<td>Z1</td>
<td>✔</td>
</tr>
<tr>
<td>MicaZ</td>
<td>✔</td>
</tr>
<tr>
<td>IRIS Mote</td>
<td>✔</td>
</tr>
<tr>
<td>BTN Not</td>
<td>✔</td>
</tr>
<tr>
<td>Mica2</td>
<td>✔</td>
</tr>
<tr>
<td>TinyNode</td>
<td>✔</td>
</tr>
<tr>
<td>Wismote</td>
<td></td>
</tr>
<tr>
<td>ESB</td>
<td></td>
</tr>
<tr>
<td>Sky</td>
<td></td>
</tr>
<tr>
<td><strong>Simulated platform</strong></td>
<td></td>
</tr>
<tr>
<td>TinyOS mote</td>
<td>✔</td>
</tr>
<tr>
<td>Contiki mote</td>
<td></td>
</tr>
</tbody>
</table>

Table 11.2 Selected mote platforms for portability evaluation
The benchmark applications are selected so that they cover basic functions of a sensor node such as sensing, scheduling and communication. The list of benchmark applications is presented in Table 11.3.

<table>
<thead>
<tr>
<th>Application</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Sensor</td>
<td>Reads the data from temperature sensor at the rate 1 Hz and prints data (for debugging).</td>
</tr>
<tr>
<td>Read Sensor and Blink</td>
<td>Reads the data from temperature sensor at the rate 1 Hz and toggles the red led when the temperature value exceeds a threshold.</td>
</tr>
<tr>
<td>Broadcasting</td>
<td>Creates a message and sends it to mote’s neighbors.</td>
</tr>
<tr>
<td>Periodic Broadcasting</td>
<td>Creates a message periodically at the rate 1 Hz and sends them to mote’s neighbors.</td>
</tr>
<tr>
<td>Multi-hop Sending</td>
<td>Creates a message periodically at the rate 1 Hz and sends it to mote’s neighbors. Every time, a mote receives a message from its neighbors, it will continue forwarding the message to the network.</td>
</tr>
</tbody>
</table>

Table 11.3 Benchmark applications for portability evaluation

11.2.2 Portability evaluation result

![Diagram](image)

**Figure 11.3 The benchmark applications evaluation process**

Figure 11.3 illustrates the evaluation process for the benchmark applications. As shown in the Figure 11.3 (a), the benchmark applications are defined once in terms of PIM at a high-level of abstraction and then source codes and binary codes for all target platforms
are automatically generated and compiled thanks to the MDD framework. This process is transparent to the application developers, and require none or slight modification by the developers to complete.

Figure 11.3 (b) illustrates the situation that all the benchmark applications must be ad-hoc programmed and compiled for each combination of operating system and mote hardware platform. In this case the applications are developed completely platform-specific at a low level of abstraction. Additionally, for each combination of operating system and mote hardware the application source code may also differ.

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read Sensor</td>
</tr>
<tr>
<td>Z1 + TinyOS</td>
<td>+</td>
</tr>
<tr>
<td>Z1 + Contiki</td>
<td>+</td>
</tr>
<tr>
<td>MicaZ + TinyOS</td>
<td>+</td>
</tr>
<tr>
<td>MicaZ + Contiki</td>
<td>+</td>
</tr>
<tr>
<td>IRIS mote + TinyOS</td>
<td>*</td>
</tr>
<tr>
<td>IRIS mote + Contiki</td>
<td>*</td>
</tr>
<tr>
<td>BTN ode + TinyOS</td>
<td>*</td>
</tr>
<tr>
<td>TinyNode + TinyOS</td>
<td>+</td>
</tr>
<tr>
<td>Wismote + Contiki</td>
<td>+</td>
</tr>
<tr>
<td>ESB + Contiki</td>
<td>+</td>
</tr>
<tr>
<td>Sky + Contiki</td>
<td>+</td>
</tr>
<tr>
<td>Simulated TinyOS</td>
<td>+</td>
</tr>
</tbody>
</table>

+ : Binary code is generated correctly without any manual modification from developer
* : Binary code is generated but manual turning on the source code is needed to have a functional application
- : Binary code is not generated automatically

Table 11.4 Portability evaluation result

Table 11.4 shows the evaluation result. In the table, symbol “+” means the binary code of a benchmark application for a target platform is generated automatically and the application works as expected without any additional manual modification from
developers. The “*” notation means the binary code is generated but the source code must be tuned manually in order to have a working application. In most cases, the turning is to change some platform related parameters that are defined in the source code header file. The “-” denotes that the generated source code (in C or NesC) has some compiler errors and the binary code of the application is not generated. In this case, the developers must spend some time and effort on investigation and debugging. This situation often occurs with complex benchmark applications like Multi-hop Sending or Periodic Broadcasting.

As presented in the Table 11.4, the framework is able to successfully generate binary codes for a benchmark application in 81% of all the test cases, without or with a slight manual modification in the source code. Specifically, the code generation is completely automatic in 40% of the total test cases, and in 41% of all the experiments a simple source code turning is necessary to result a working binary code. This situation where the binary code generation is impossible occurs in 18% of all the test cases, mostly for complex applications.

Figure 11.4 summarizes the portability evaluation result. Clearly, the result shows that our framework is able to abstract away the platforms low-level details and increase the application portability.

11.3 Framework overhead evaluation

11.3.1 Overhead evaluation metrics

The main kind of overhead imposed by the MDD framework over the original benchmark applications is memory usage of the applications. Due to the hardware
limitation in WSN, this overhead is particularly important. The memory usage of an application is usually measured by two metrics: memory footprint and binary image size of the application. The memory footprint of an application refers to the amount of RAM memory that the application uses when running, while the size of the application’s binary image represents the amount of ROM memory needed to upload the application to the mote memory during installation.

The footprint and image size of an application depends on the application, the operating system and the microcontroller of the target mote. Both metrics can be computed after compiling the application’s source code. For each microcontroller, developers may use specific utilities to calculate the metric values. For instance, `msp430-size` utility is used for MSP430 microcontroller (Z1 or Sky mote), and `avr-size` is used for Atmel microcontroller (MicaZ or IRIS mote), respectively. When executing, these commands will report the memory layout and size in bytes for each segment of the target program. Listing 11.1 shows the result when the command `msp430-size build/z1/main.exe` is executed, where `main.exe` is a compiled program for Z1 mote with TinyOS.

<table>
<thead>
<tr>
<th>Memory layout of a program on Z1 mote running TinyOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
</tr>
<tr>
<td>6590</td>
</tr>
</tbody>
</table>

Listing 11.1 Application memory layout returned by msp430-size utility

In the above listing, the `text` segment contains the executable instructions of the program. In a WSN mote, this segment is usually placed at ROM memory. The `data` segment is used for the initialized data such as global or static variables in the program. This segment is also placed at ROM memory of a mote; furthermore, a copy of the segment is placed at RAM memory after starting the program. The program’s uninitialized variables are stored in the `bss` segment. This segment is also placed in the RAM of the WSN mote. The `dec` and `hex` is total size of the `text`, `data` and `bss` segment in decimal and hexadecimal respectively. From the explanation above, we can compute the footprint and the image size of an application as the following:

- The memory footprint (RAM consumption) is computed as the sum of `data` segment and `bss` segment.
- The binary image size (ROM consumption) is computed as the sum of `text` segment and `data` segment.

For every application in the benchmark set, on one hand we wrote the source code from scratch for any target platform (a possible combination of mote operating system and hardware). On the other hand, we used the framework to create the application for
11.3 Framework overhead evaluation

the target platforms. As mentioned in the previous section, for some applications, a manual turning/programming must be performed on the generated source code to result a working binary image of the application.

In both development processes, we recorded the memory footprint and binary image size for each benchmark application with every target platform for later analysis. Table 11.5 presents a part of our data: the footprint and image size of Read Sensor application that is manually developed for six different target platforms. The evaluation result is discussed in detail in the subsequence section.

<table>
<thead>
<tr>
<th>Platform</th>
<th>TinyOS</th>
<th>ContikiOS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Footprint</td>
<td>Image size</td>
</tr>
<tr>
<td>Z1</td>
<td>137 bytes</td>
<td>6598 bytes</td>
</tr>
<tr>
<td>MicaZ</td>
<td>47 bytes</td>
<td>2862 bytes</td>
</tr>
<tr>
<td>IRIS mote</td>
<td>47 bytes</td>
<td>3126 bytes</td>
</tr>
</tbody>
</table>

Table 11.5 An example of memory footprint and image size of Read Sensor application

11.3.2 Overhead evaluation result

Every application in the benchmark set is developed manually and automatically for 14 various target platforms. The stacked column chars in Figure 11.6 presents memory footprints and binary image size of the first application in the benchmark set, the Read Sensor application. The memory footprint and binary image size of the Read Sensor applications are relatively small. If the target operating system is TinyOS, the application has a footprint of few hundred bytes and an image size of few kilobytes. If the target operating system is Contiki, the memory footprint and binary image size of Read Sensor application increase to around two kilobytes RAM and thirty kilobytes ROM. Contiki applications need more RAM and ROM than TinyOS application because when compiling a Contiki application the code for network communication stack is also compiled even if the application does not need it. In case of TinyOS application, the compiler only compiles components that are used by the application. As shown in Figure 11.6, the size of generated binary code of Read Sensor application is bigger than the size of manually programmed application for all target platforms. It means that the framework really introduces an amount of RAM and ROM memory overhead. This overhead varies based on the operating system and the hardware platform. For TinyOS Read Sensor application, the overhead is around 180 bytes on RAM and 1500 bytes on ROM. For Contiki, the overhead is bigger, around 1100 bytes on RAM and 4200 bytes on ROM respectively.
Figure 11.5 Comparison of memory footprints and image sizes between Read Sensor and Blink applications

Figure 11.6 Comparison of memory footprints and image sizes between Read Sensor applications
11.3 Framework overhead evaluation

Figure 11.5 depicts the comparison of memory footprint and binary image size between generated Read Sensor and Blink application and the manually programmed ones for different target platforms. Although the RAM and ROM usage is different from the case of Read Sensor application, we can observe the same amount of memory overhead introduced by our framework on the application for different target platforms. For instance, the RAM overhead is around 160 bytes for TinyOS and around 1100 bytes for Contiki, whereas, the ROM overhead is around 1500 bytes and 4200 bytes for TinyOS and Contiki respectively. The same situation occurs with other applications like Broadcasting and Periodic Broadcasting, as shown in Figure 11.7 and Figure 11.8. For the most complex application in the benchmark set, the Multi-hop Sending application, the memory overheads are bigger, as illustrated in Figure 11.9. Specifically, the RAM overhead for TinyOS and Contiki is around 320 bytes and 1200 bytes, while the ROM overhead is around 2200 bytes for TinyOS and 4500 bytes for Contiki.

Finally, Figure 11.10 - 13 presents the detailed comparison of memory overhead data introduced by our framework on all benchmark applications for every platform in the experiment.
Chapter 11 Framework evaluation

Figure 11.8 Comparison of memory footprints and image sizes between Periodic Broadcasting applications

Figure 11.9 Comparison of memory footprints and image sizes between Multi-hop Sending applications
11.3 Framework overhead evaluation

Figure 11.10 Memory footprint overhead introduced by the framework on benchmark applications for TinyOS

Figure 11.11 Memory footprint overhead introduced by the framework on benchmark applications for Contiki
Chapter 11 Framework evaluation

Figure 11.12 Binary image size overhead introduced by the framework on benchmark applications for TinyOS

Figure 11.13 Binary image size overhead introduced by the framework on benchmark applications for Contiki
12 Design and optimize a real-world application

This chapter discusses the design, implementation and optimization of a real-world application named TempSense, where a sensor network is used to monitor the temperature in an area. In our experiment, we designed and implemented TempSense on a target platform (Z1 and ContikiOS) using the MDD framework. Then we optimized the application in terms of power efficiency and reliability with the two proposed optimization approaches SSDoE and MOEA.

Our results demonstrate that the application non-functional constrains such as power consumption and reliability can be improved during the optimization process by selecting a proper set of adjustable parameters instead of the default values chosen by developers. We observed that both approaches SSDoE and MOEA are able to achieve a good approximation of the Pareto set for the multi-objective optimization problem described in Section 10.1.2. The Pareto frontier allows developers to manage the tradeoff between power efficiency and reliability when choosing a good set of input adjustable parameters. In Section 12.1, we describe TempSense application, as well as its input parameters and non-functional constrains as optimization objectives. Then the optimization results using SSDoE and MOEA are presented in Section 12.2 and Section 12.3 respectively.

12.1 TempSense application

TempSense application falls into the category of monitoring applications where a sensor network is used to collect temperature data in an area. Each node in the WSN is equipped with a sensor to periodically sense the local temperature once per second, and forwards the data along the network to a base station. An improved version of gossip routing protocol [77] is used for data transmission from nodes to the base station in the network:

- For the first \( K \) hops, messages are forwarded with the probability 1.
- From the hop \( K+1 \), messages are forwarded with the probability \( P \).
- Message lifetime is implemented to void infinite loop. A message will be dropped if the lifetime is exceeded a threshold \( L \).

Figure 12.1 shows a PIM model that is created for TempSense application using the proposed framework. In the PIM, a rule set that consists of three rules is defined to describe the behaviors of the sensor nodes. After a time \( T_s \) (for example \( T_s=2s \)) the rule set is applied to all message in the buffer. The first rule is used to drop all messages that have exceeded their lifetime. The second rule is used to forward every message that has
a hop count smaller than $K+1$. The remaining messages in the buffer are gossiped with the probability $P$ using the third rule.

When creating the PIM model, developers can assign necessary values for the adjustable parameters like $BufferLength$ or $MaxLifeTime$ to define the maximum length of the message buffer or the maximum lifetime of messages. The default value and the value range of each parameter are also defined by setting value of the $Defaultvalue$, $Minvalue$ and $Maxvalue$ attributes. In $TempSense$ application, developers often care about the reliability and the energy efficiency of the system. Therefore, non-functional properties could also be defined such as energy per delivered packet ($NFP1$) and packet loss ratio ($NFP2$). The developer may want to optimize the application in term of energy efficiency and reliability, so he may define the non-functional constrains as $Min(NFP1)$ $\&\&$ $Min(NFP2)$. Table 12.1 presents the list of adjustable parameters and non-functional constrains for $TempSense$ application. The information will be used later in the optimization process.
Table 12.1 List of adjustable parameters and non-functional constrains for TempSense application

<table>
<thead>
<tr>
<th>Name</th>
<th>Describe</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFP1</td>
<td>Energy per delivered packet</td>
</tr>
<tr>
<td>NFP2</td>
<td>Packet loss ratio</td>
</tr>
<tr>
<td>Non-functional</td>
<td>Min(NFP1) &amp;&amp; Min (NFP2)</td>
</tr>
<tr>
<td>constrains</td>
<td></td>
</tr>
</tbody>
</table>

Now we can formulate the optimization problem for the non-functional constrains in TempSense as the following:

\[
\begin{align*}
\text{min} & \quad P(X) = [P_1(X), P_2(X)] \\
\text{subject to} & \quad X = [X_1, X_2, X_3, X_4, X_5, X_6] \\
& \quad 1 \leq X_1 \leq 30; \quad 1 \leq X_2 \leq 20; \quad 1000 \leq X_3 \leq 10000; \\
& \quad 100 \leq X_4 \leq 5000; \quad 1 \leq X_5 \leq 10; \quad 0.1 \leq X_6 \leq 1
\end{align*}
\]

Equation 12.1

To test the TempSense application, we configured a simulated network of 200 nodes which are distributed randomly in an area of 500 x 500 meters. The simulated network has a sink node that receives temperature data from all nodes in the network. Other nodes are implemented with TempSense application to sense and send temperature data. The radius of the transmission range and the interference range of each node in the simulation are 50 meters and 80 meters respectively. In each experiment, we run the simulation three times, with each round being one minute long. The simulator collects all the non-functional metrics of the network after each simulation round but only the average metric values are reported to the optimization component.
12.2 Optimizing TempSense with SSDOE

We use SSDOE approach presented in Section 10.3.1 to optimize TempSense application based on the non-functional constrains. This optimization process is described as the following:

- In step 1, we selected three representative values for each adjustable parameter. Thus, the first sample set of the solution space contains: $3^6 = 729$ points. For each point in the sample set, we run the simulation to measure the corresponding non-functional metric values including energy per delivered packet and packet loss ratio. The pair of two metric values forms a point in the objective space.
- In step 2, with the finite set of sample points in the solution space and the corresponding points in the objective space, we approximate the relationship between two non-functional metrics and the input adjustable parameters. We use a linear statistical regression model to do so. The input and output values are fitted to create the statistical model:

$$P_1 = \beta_0 + \sum_{i=1}^{6} \beta_i X_i + \varepsilon$$
$$P_2 = \beta_0 + \sum_{i=1}^{6} \beta_i X_i + \varepsilon$$

Table 12.2 presents the statistical model obtained after the second step.

<table>
<thead>
<tr>
<th>Sample points in solution space</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
<th>X5</th>
<th>X6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value 1</td>
<td>0.03333</td>
<td>0.05</td>
<td>0.1</td>
<td>0.02</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Value 2</td>
<td>0.66667</td>
<td>0.5</td>
<td>0.2</td>
<td>0.5</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>Value 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistical models</th>
<th>$\beta_0$</th>
<th>$\beta_1$</th>
<th>$\beta_2$</th>
<th>$\beta_3$</th>
<th>$\beta_4$</th>
<th>$\beta_5$</th>
<th>$\beta_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.16322</td>
<td>0.18235</td>
<td>0.09215</td>
<td>0.25816</td>
<td>0.03718</td>
<td>0.12982</td>
<td>0.10922</td>
</tr>
<tr>
<td>P2</td>
<td>0.24924</td>
<td>0.29655</td>
<td>-0.07006</td>
<td>0.38189</td>
<td>-0.18976</td>
<td>0.05572</td>
<td>-0.05973</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1 regression statistic</th>
<th>R Square</th>
<th>Standard error</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.92</td>
<td>0.04</td>
<td>729</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2 regression statistic</th>
<th>R Square</th>
<th>Standard error</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.90</td>
<td>0.07</td>
<td>729</td>
</tr>
</tbody>
</table>

Table 12.2 Obtained statistical models to predict two non-functional metrics from input parameters in TempSense
• In step 3 we increase the number of sample points in the solution space by selecting ten representative values for each adjustable parameter. Thus, the sample set of the solution space now contains: $10^6 = 1,000,000$ points. For each point in the sample set, we predict the corresponding non-functional metrics using the statistical model obtained after the second step. The result is a set of one million points in the objective space. Figure 12.2 shows the estimated objective space with one million points.

![Figure 12.2 The objective space with one million points when optimizing TempSense](image)

• Finally in step 4, we apply the algorithm presented in Listing 10.2 to compute all points in the Pareto frontier. As shown in Figure 12.3, we find 47 points in the Pareto frontier and user can choose a proper solution by considering the trade-off between two non-functional metrics. Table 12.3 presents an example set of ten solutions selected from the Pareto frontier. Clearly, the value sets are better than the default values of the adjustable parameters chosen by developers in term of power consumption and reliability of the application.
12.2 Optimizing TempSense with SSDOE

<table>
<thead>
<tr>
<th></th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
<th>X5</th>
<th>X6</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>20</td>
<td>10</td>
<td>2000</td>
<td>2500</td>
<td>5</td>
<td>0.6</td>
<td>0.49007</td>
<td>0.41695</td>
</tr>
<tr>
<td>Solution 1</td>
<td>5</td>
<td>8</td>
<td>3000</td>
<td>1500</td>
<td>3</td>
<td>0.6</td>
<td>0.26081</td>
<td>0.10365</td>
</tr>
<tr>
<td>Solution 2</td>
<td>5</td>
<td>10</td>
<td>3000</td>
<td>1000</td>
<td>3</td>
<td>0.5</td>
<td>0.26542</td>
<td>0.10014</td>
</tr>
<tr>
<td>Solution 3</td>
<td>5</td>
<td>8</td>
<td>3000</td>
<td>1000</td>
<td>3</td>
<td>0.4</td>
<td>0.25783</td>
<td>0.11883</td>
</tr>
<tr>
<td>Solution 4</td>
<td>10</td>
<td>8</td>
<td>2000</td>
<td>2500</td>
<td>3</td>
<td>0.3</td>
<td>0.23850</td>
<td>0.21751</td>
</tr>
<tr>
<td>Solution 5</td>
<td>10</td>
<td>6</td>
<td>2000</td>
<td>2000</td>
<td>3</td>
<td>0.5</td>
<td>0.23478</td>
<td>0.23648</td>
</tr>
<tr>
<td>Solution 6</td>
<td>10</td>
<td>4</td>
<td>1000</td>
<td>1500</td>
<td>4</td>
<td>0.6</td>
<td>0.33163</td>
<td>0.05214</td>
</tr>
<tr>
<td>Solution 7</td>
<td>5</td>
<td>6</td>
<td>2000</td>
<td>1000</td>
<td>3</td>
<td>0.5</td>
<td>0.34835</td>
<td>0.03709</td>
</tr>
<tr>
<td>Solution 8</td>
<td>5</td>
<td>6</td>
<td>2000</td>
<td>1500</td>
<td>3</td>
<td>0.5</td>
<td>0.35638</td>
<td>0.03565</td>
</tr>
<tr>
<td>Solution 9</td>
<td>15</td>
<td>8</td>
<td>3000</td>
<td>1500</td>
<td>4</td>
<td>0.8</td>
<td>0.43283</td>
<td>0.01106</td>
</tr>
<tr>
<td>Solution 10</td>
<td>15</td>
<td>8</td>
<td>3000</td>
<td>1000</td>
<td>3</td>
<td>0.9</td>
<td>0.43573</td>
<td>0.01105</td>
</tr>
</tbody>
</table>

Table 12.3 An example set of ten solutions obtained when optimizing TempSense application using SSDOE

Figure 12.3 The Pareto frontier obtained when optimizing TempSense application using SSDOE
12.3 Optimizing TempSense with MOEA

MOEA approach with SPEA2 is used to solve the optimization problem as described in Section 10.3.2. We define the population size of the working population $N = 50$ individuals. In our problem, a solution candidate is composed of the set of real numbers corresponding to the set of adjustable parameter values. Therefore, we used a real-coded method where an individual (a solution in the solution space) is a vector of floating point numbers. In order to generate new solutions from existing ones, binary tournament selection with replacement, Simulated Binary Crossover (SBX) and polynomial mutation operators are implemented. The fitness of each solution candidate is evaluated using the simulator. Our tests show that the convergence occurs quickly within 50 generations. Figure 12.4 shows the Pareto frontier obtained after the optimization process. Figure 12.5 presents the comparison between the Pareto frontiers achieved using two approaches SSDOE and SPEA2.

![Figure 12.4 The Pareto frontier obtained when optimizing TempSense application using SPEA2](image)
Figure 12.5 Comparison between two approaches SSDOE and SPEA2
Chapter 13 Discussion and Conclusion

13 Discussion and Conclusion

WSNs differ from conventional distributed systems in many aspects. As sensor applications become more complex and diverse, rising abstraction level in designing of WSNs software and tailoring application for non-functional constraints becomes a more important and challenging task. In order to deal with this problem, support is needed in designing, implementing and optimizing of WSNs software. We consider our work in the thesis as one step forward towards the direction of providing such support. In this chapter, we summarize the major contributions obtained from the thesis work. Firstly we discuss how the main objectives are addressed by the propose solution. Then the main contributions of the thesis work are reviewed. Finally, we identify some possible future work and interesting research directions.

13.1 Requirement revisited

Different aspects of the challenge of developing a WSN application are studied in this dissertation as show in Section 1.2.1. The identified challenges and key characteristics directly translate to a number of requirements presented in Section 1.2.2. The main objective of the thesis work is to provide a comprehensive approach fulfilling all requirements at the same time. Thus, in this section we review how each requirement is addressed, as well as how the objectives are achieved by the propose solution.

1. Design a multi-layered software architecture clearly distinguishing the different abstraction levels

An overview of the software architecture is presented in Chapter 6. The architecture aims to abstract away the hardware heterogeneity to support writing applications independently of the platform. It is designed in a multi-layered fashion, which allows focusing on details of each architectural component. At the application layer, a programming model and a Domain-Specific Language (DSL) has been created to support the development of portable applications on top of the previous layer (refer to Chapter 5 & 7). The DSL also enable the modeling of the application-specific requirements as the non-functional resource constraints. An abstraction layer for heterogeneous WSNs operating systems are introduced to provide a standardized way to access to the underlying architecture. At the operating system level, several operating systems for sensor nodes are considered and analyzed including TinyOS (versions 2.x) and Contiki. At the hardware level, a reasonable set of physical devices have to be studied in order to elaborate a generic and flexible hardware model,
including the analysis of the resources and their properties, functionalities, and services.

2. **Design and implement a framework that adopts the architecture above**

   The framework is designed as a plug-in of the popular Eclipse Integrated Development Environment. The framework provides developers with a graphical interface to develop WSNs software (refer to Section 7.3). With each considered operation system, a translator component and a compiler are integrated in order to interpret and transform the DSL into a lower level language and generate the program code (refer to Chapter 8). Furthermore, the framework includes a simulation component (Chapter 9) that allows the developer to simulate the deployment of the software.

3. **Define a reusable methodology for addressing the problem of optimizing non-functional constraints**

   In Section 7.1 we identify, capture and represent major WSN application’s specific requirements as the non-functional constraints that need to be optimized. After that, in Chapter 10, we analyze the optimization problem and discuss two approaches for the problem namely SSDOE and MOEA. The SSDOE approach could be considered as a broad but shallow search method which explores the search space in a systematic and even manner. In contrary, the MOEA could be considered as a narrow but deep heuristic search method which explores the search space in an uneven manner. An optimization component based on SPEA2 is integrated into the framework to support developers to optimize their applications in terms of non-functional constraints.

4. **Demonstrate several test cases and evaluating the performance of the approach**

   Chapter 11 presents the evaluation of our development framework. We present different scenarios and conduct some experiments to assess the ease at which novice and intermediate users can develop correct and power efficient WSN applications, the portability level achieved by developing applications at a high-level of abstraction, and the overhead due to the framework usage in term of the footprint and executable code size of the application. In Chapter 12, we discuss the design, implementation and optimization of TempSense, a real-world application, where a sensor network is used to monitor the temperature in an area. In this experiment, we design and implement TempSense on a target platform (Z1 and ContikiOS) using the MDD framework. Then we optimize the application in terms of power efficiency and reliability with two proposed approaches SSDOE and MOEA.
From the discussion above, we can conclude that the solution proposed in this thesis is able to address all challenges and objectives identified in Section 1.2.1 and 1.2.2 respectively. Thus, it is justified to claim that this thesis work provides a comprehensive approach to WSN application development in order to improve the reusability, flexibility, and maintainability of the software.

13.2 Contribution

The research work presented in this thesis leads to a number of contributions to the state of the art of software development in wireless sensor networks. The thesis work proposes the formalization of the sensor node’s software architecture equipped with a completed framework for developing and optimizing WSN software. The proposed approach reduces the effort of learning and developing associated with the heterogeneity and complexity exhibited by sensor nodes and WSN operating systems. Although the approach is designed to be applicable for WSN, in particular contributions have been made in the area of embedded and mobile system or mobile robot. These contributions can be summarized as follows:

- **The sensor node software architecture**: The description of the architecture has fulfilled the MDA standard principles, which encourage defining independent models of the systems, including a transformation process to allow obtaining platform specific models. The design of the architecture has been accomplished in a multi-layered approach, where each layer interacts through a well-defined interface. Additionally, physical devices are described through XML manifests and schemas, which allow the contents specification to be decoupled from the instantiation, and hide the complexity of the underlying platform for programmers.

- **An appropriate method for representing, measuring and optimizing non-functional constraints**: In the context of software development, non-functional aspects play an important role and should be analyzed and modeled as early as possible in the development cycle. In this thesis, we discuss modeling, measuring and optimizing of non-functional constraints of WSN applications. Our contribution is threefold. First, we classify non-functional properties into three classes (qualitative, inferable and measurable properties), and we discuss modeling of these non-functional constraints using our proposed DSL. Second, we provide suitable measurement and configuration techniques for each class of non-functional properties. Last but not least, we implement two different techniques to analyze and optimize the application in terms of evaluating the tradeoff between different constraint criteria.
13.3 Future work

- **A Domain Specific Language to describe applications on top of the architecture:** At the highest abstraction level in the proposed architecture, a DSL to describe WSN applications has been designed. It has been elaborated to uncouple the programming language and its execution model used by the underlying OS from the application programming.

- **A completed framework for WSN applications development:** A graphical framework for developing WSN applications using a visual notation is presented in this thesis. This development framework constitutes a complete IDE and provides a semi-automatic support for the WSN applications life cycle, using the proposed architecture. Several outputs are generated: high-level code (in DSL), operating system-specific source files and the final executable code for the target sensor node.

**13.3 Future work**

There still are a number of open questions which are subject of future work, and our work could be further extended and improved in many different aspects. In the following paragraphs we discuss the most important ones with respect to the main building blocks of our solution approach:

1. **Regarding to the DSL and programming model**
   The rule-based programming model is simple and effective when expressing the local behavior of sensor nodes in a network. However, our rule systems could be extended in the future to support various kinds of WSN applications. For instance, an application for mobile sensor network that requires complex in-network aggregation activity presented in [134], or an application for sensor nodes in the sensor and actuator network [44]. Besides, the DSL could be extended to represent different component of these new kinds of applications.

2. **Regarding to the framework and the software architecture**
   Currently, our framework supports two mote’s operating systems TinyOS and Contiki, and a limited number of hardware platforms. In the future, the core components and libraries could be improved to support more operating systems and hardware platform.

3. **Regarding to the non-functional properties of WSN application**
   We classify non-functional properties into three different classes: qualitative properties, inferable properties and measureable properties. At the moment, only measureable properties are supported in our solution. Dealing with other class of non-functional properties still remains for future work.
4. **Regarding to the optimization process**

The high cost of experiment time remains an open problem in the optimization process. This problem should be addressed in the future. In order to reduce the total runtime of the optimization task, the following ideals could be considered:

- All simulations are independent and can be executed in parallel if sufficient processing hosts are available.
- Before the optimization, we may investigate the relationship between the input adjustable parameters and the output non-functional metrics in detail to identify which parameters are the best predictors of the output metrics. A statistical model should be built to support doing so. Then it is possible to keep only the significant parameters while safely discarding others to reduce the problem size.
- A surrogate-assisted MOEA [152] could be a promising approach for the optimization problem presented in the thesis. The surrogate-assisted algorithms are currently the state-of-the-art MOEA. They are often used in the expensive optimization problems that introduce a high cost of evaluating fitness values for each candidate solution.
Part IV
Appendices
## Appendix A  
### WSN technologies

#### A1  Mote hardware platform evolution

<table>
<thead>
<tr>
<th>Platform</th>
<th>Year</th>
<th>Microcontroller</th>
<th>Memory</th>
<th>Serial comm.</th>
<th>Wireless com.</th>
<th>Programming &amp; OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mica2</td>
<td>2002</td>
<td>ATMega 128L 8 bit</td>
<td>128K ROM 4K RAM 512K Flash</td>
<td>51-pin UART</td>
<td>Chipcon CC1000 868/916 MHz</td>
<td>nesC - TinyOS, SOS and MantisOS support</td>
</tr>
<tr>
<td>BTnode</td>
<td>2003</td>
<td>ATMega 128L 8 bit</td>
<td>128K ROM 4K RAM 4K EPROM</td>
<td>51-pin UART</td>
<td>Chipcon CC1000 868/916 MHz</td>
<td>C &amp; nesC NutOS &amp; TinyOS support</td>
</tr>
<tr>
<td>MicaZ</td>
<td>2004</td>
<td>ATMega 128L 8 bit</td>
<td>128K ROM 4K RAM 512K Flash</td>
<td>51-pin UART</td>
<td>TI CC2420 802.15.4/ ZigBee compliant</td>
<td>nesC &amp; Tiny OS support</td>
</tr>
<tr>
<td>Telos B</td>
<td>2005</td>
<td>TI MSP430 16 bit</td>
<td>60K ROM 10K RAM 48K Flash</td>
<td>10-pin RS-232</td>
<td>TI CC2420 802.15.4/ ZigBee compliant</td>
<td>C &amp; nesC Contiki, TinyOS, SOS and MantisOS Support</td>
</tr>
<tr>
<td>Eyes</td>
<td>2006</td>
<td>TI MSP430 16 bit</td>
<td>60K ROM 2K RAM 4K EPROM</td>
<td>TR1001 UART/GP 48 MHz</td>
<td>TI CC2420 802.15.4/ ZigBee compliant</td>
<td>C PEER OS support</td>
</tr>
<tr>
<td>Imote 2</td>
<td>2007</td>
<td>Intel PXA271 16 bit</td>
<td>32M Flash 32M RAM</td>
<td>USB/GP 802.15.4</td>
<td>TI CC2420 802.15.4/ ZigBee compliant</td>
<td>C &amp; nesC Microsoft .NET Micro, Linux, TinyOS Support</td>
</tr>
<tr>
<td>SHIMMER</td>
<td>2008</td>
<td>MSP 430 16 bit</td>
<td>48K Flash 10K RAM microSD support</td>
<td>UART/I2C</td>
<td>Jemini JN5148 802.15.4/ ZigBee compliant</td>
<td>nesC Tiny OS support</td>
</tr>
<tr>
<td>iSense 2</td>
<td>2009</td>
<td>RISC 32 bit</td>
<td>512K Flash 128K RAM</td>
<td>UART/I2C</td>
<td>JniCC 802.15.4/ ZigBee compliant</td>
<td>C++ modular operating and networking firmware</td>
</tr>
<tr>
<td>Arduino BT</td>
<td>2010</td>
<td>ATMega 328 8 bit</td>
<td>32K Flash 2K SRAM 1K ROM</td>
<td>UART/PWM/I2C</td>
<td>Bluetooth</td>
<td>Arduino language is based on C/C++</td>
</tr>
<tr>
<td>Zolertia Z1</td>
<td>2011</td>
<td>MSP 430 16 bit</td>
<td>96K Flash 8K RAM</td>
<td>UART/I2C</td>
<td>Chipcon CC2420 802.15.4/ ZigBee compliant</td>
<td>C, nesC Contiki &amp; TinyOS support</td>
</tr>
<tr>
<td>WaspMote Pro</td>
<td>2012</td>
<td>ATMega 1281 8 bit</td>
<td>128K Flash 8K RAM 4K ROM microSD support</td>
<td>UART/I2C/USB/SPI</td>
<td>3G/GPRS/802.15.4/ ZigBee/Bluetooth/RFID/NFC</td>
<td>C with WaspMote IDE Wasp firmware</td>
</tr>
</tbody>
</table>

Table A.1 Mote hardware platform evolution from 2002 to 2012
### A2 Physical layer design requirements

<table>
<thead>
<tr>
<th>Design requirements</th>
<th>Solutions</th>
<th>Main characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bandwidth choices</strong></td>
<td>Narrow band, Spread-spectrum, Ultra-wideband</td>
<td>Spread-spectrum is preferred over narrow band because of its ability to reduce power, communicate effectively, and more robustness to interference and multi-channel impairment. Ultra-wideband is an alternate solution to spread-spectrum.</td>
</tr>
<tr>
<td><strong>Radio architecture</strong></td>
<td>FN frequency synthesizer with ΣΔ modulator, WiseNet</td>
<td>Fast startup radio architectures minimize both energy and time. FN frequency synthesizer with ΣΔ modulator achieves fast startup time and data rate by adjusting the loop bandwidth. WiseNet achieves low energy consumption by using a duty-cycled radio with a low power MAC protocol.</td>
</tr>
<tr>
<td><strong>Modulation Schemes</strong></td>
<td>Binary modulation, M-FSK modulation, M-PSK modulation, M-QAM modulation</td>
<td>Multi-level modulation achieves more energy efficiency than binary modulation when the startup time is short. M-FSK is more efficient compared to M-PSK and M-QAM when M &gt; 8.</td>
</tr>
</tbody>
</table>

**Table A.2 Overview of physical layer design requirements in WSN**

### A3 MAC protocol analysis

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S-MAC</strong></td>
<td>Time-slotted contention based</td>
<td>No</td>
<td>CSMA/CA</td>
<td>Node uses a listen/sleep schedule which requires synchronization among neighboring nodes</td>
<td>Reduce the energy waste caused by idle listening</td>
</tr>
<tr>
<td><strong>TRAMA</strong></td>
<td>Time-slotted random and scheduled access</td>
<td>Yes</td>
<td>TDMA</td>
<td>Achieves adequate throughput and fairness through transmitter election algorithm and channel reuse</td>
<td>Schedule sleep intervals and turn radio off when idle, collision avoidance scheduling</td>
</tr>
<tr>
<td><strong>B-MAC</strong></td>
<td>Clear channel assessment (CCA)</td>
<td>No</td>
<td>CSMA</td>
<td>Bi-directional interface for reconfiguration of system services to optimize performance</td>
<td>Low power listening (LPL) time for energy efficiency</td>
</tr>
<tr>
<td><strong>Z-MAC</strong></td>
<td>Time-slotted random and scheduled access</td>
<td>Yes</td>
<td>TDMA/CSMA</td>
<td>Combine the strengths of TDMA and CSMA while reducing their weaknesses</td>
<td>Use LPL feature from B-MAC</td>
</tr>
<tr>
<td><strong>LPR-MAC</strong></td>
<td>Time-slotted contention based slot reservation</td>
<td>Yes</td>
<td>TDMA</td>
<td>Increases the probability of success in packet transmission by adapting to traffic requirements to maximize data</td>
<td>Nodes sleep and wake up based on assigned data slot</td>
</tr>
</tbody>
</table>
Table A.3 Overview of the representative MAC protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>GOSSIP</th>
<th>DSDV</th>
<th>AODV</th>
<th>Directed Diffusion</th>
<th>LEACH</th>
<th>GAF</th>
<th>SPEED</th>
<th>SEC-ROU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification</td>
<td>Flooding</td>
<td>Pro-active</td>
<td>Re-active</td>
<td>Data-centric</td>
<td>Hierarchical</td>
<td>Geographical based</td>
<td>Real-time</td>
<td>Hierarchical</td>
</tr>
<tr>
<td>Position awareness</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Negotiation based</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Localization based</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Query-based</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Power usage</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Fair</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Data aggregation</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>QoS support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scalability</td>
<td>Low</td>
<td>Low</td>
<td>Fair</td>
<td>Fair</td>
<td>Good</td>
<td>Good</td>
<td>Fair</td>
<td>Good</td>
</tr>
<tr>
<td>Data security</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table A.4 Overview of the representative routing protocols

A4 Routing protocol analysis

A5 Transport protocol analysis
### A6 Mote’s operating system analysis

<table>
<thead>
<tr>
<th>Feature</th>
<th>TinyOS</th>
<th>Contiki</th>
<th>Mantis OS</th>
<th>LiteOS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture</strong></td>
<td>Monolithic</td>
<td>Modular</td>
<td>Layered</td>
<td>Modular</td>
</tr>
<tr>
<td><strong>Programming model</strong></td>
<td>Event-driven</td>
<td>Proto-thread</td>
<td>Multi-threading</td>
<td>Multi-threading</td>
</tr>
<tr>
<td><strong>Scheduling</strong></td>
<td>FIFO</td>
<td>Events are fired as they occur.</td>
<td>Priority-based</td>
<td>Priority-based</td>
</tr>
<tr>
<td><strong>Memory management</strong></td>
<td>Static</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td><strong>Memory protection</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Communication protocol support</strong></td>
<td>Active message</td>
<td>uIP and Rime</td>
<td>COMM layer and Networking layer</td>
<td>File based communication</td>
</tr>
<tr>
<td><strong>Support for real-time app.</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Communication security</strong></td>
<td>TinySec</td>
<td>ContikiSec</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>File system</strong></td>
<td>Single level</td>
<td>Coffee file system</td>
<td>No</td>
<td>LiteFS</td>
</tr>
<tr>
<td>Simulation support</td>
<td>TOSSIM</td>
<td>Cooja</td>
<td>AVRORA</td>
<td>AVRORA</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Programming language</td>
<td>NesC</td>
<td>C</td>
<td>C</td>
<td>C++</td>
</tr>
</tbody>
</table>

Table A.6 Comparison of mote’s operating system
Appendix B  Software Architecture

B1  Hardware description

A DTD file to specify digital temperature sensor

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Manifest (definition, signal, assemblies, interface, resource*, constraints*)>
<!ATTLIST Manifest Signature CDATA #REQUIRED>

<!ELEMENT SENSOR (Manifest)>
<!ELEMENT action (parameter*)>
<!ATTLIST action
    name CDATA #REQUIRED
    fileAction CDATA #REQUIRED
    ReturnValueType CDATA #REQUIRED>

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
    index CDATA #REQUIRED
    type CDATA #REQUIRED>

<!ELEMENT assemblies (file)>
<!ELEMENT board EMPTY>
<!ATTLIST board
    model CDATA #REQUIRED>

<!ELEMENT constraints (#PCDATA)>
<!ELEMENT definition (model, vendor, type, sensorboard, reading)>
<!ATTLIST definition
    datasheet CDATA #REQUIRED>

<!ELEMENT export (operation)>
<!ELEMENT file EMPTY>
<!ATTLIST file
    nameAssembly CDATA #REQUIRED
    provider CDATA #REQUIRED
    fileAssembly CDATA #REQUIRED>

<!ELEMENT interface (export)>
<!ELEMENT location EMPTY>
<!ATTLIST location
    name CDATA #REQUIRED>

<!ELEMENT model (#PCDATA)>
<!ELEMENT operation (action*)>
<!ELEMENT output EMPTY>
<!ATTLIST output
    name CDATA #REQUIRED>

<!ELEMENT power EMPTY>
<!ATTLIST power
    name CDATA #REQUIRED>

<!ELEMENT reading(units, scale, voltageout, voloutvalue, tempvalue)>
<!ELEMENT voltageout EMPTY>
<!ATTLIST voltageout
    voltagein CDATA #REQUIRED
    constant CDATA #REQUIRED
    variable CDATA #REQUIRED>
```
Listing B.1 An example of temperature sensor specification
B2 Hardware Platform Specific Model of Z1 mote

Figure B.1 Hardware Platform Specific Model for Z1 mote
OS Platform Specific Model for TinyOS 2.x

Figure B.2 OS Platform Specific Model for TinyOS 2.x
B4 OS Platform Specific Model for Contiki 2.x

Figure B.3 OS Platform Specific Model for Contiki 2.x
B5  An example of the application.xml file

An XML file to describe the example PIM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<.ecore version="2.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

<package name="ExampleModel">

    <feature xsi:type="ecore:EClass" name="AppSensor">
        <feature xsi:type="ecore:EAttribute" name="SensorSamplingRate" lowerBound="1" upperBound="8" eType="ecore:EByte"/>
    </feature>

    <feature xsi:type="ecore:EClass" name="AppRule"/>

    <feature xsi:type="ecore:EClass" name="TempSensor">
        <feature xsi:type="ecore:EAttribute" name="NodeBehavior" />
    </feature>

    <feature xsi:type="ecore:EClass" name="Wireless">
        <feature xsi:type="ecore:EAttribute" name="AppComInterface" />
    </feature>

    <feature xsi:type="ecore:EClass" name="Ruleset">
        <feature xsi:type="ecore:EAttribute" name="AppNFProperty" lowerBound="-1" upperBound="1" eType="ecore:EByte"/>
        <feature xsi:type="ecore:EAttribute" name="AppRuleset" lowerBound="1" upperBound="9" eType="ecore:EString"/>
        <feature xsi:type="ecore:EAttribute" name="RulesetExecutionTime" lowerBound="1" upperBound="-1" eType="ecore:EByte"/>
        <feature xsi:type="ecore:EAttribute" name="Rule1" lowerBound="1" upperBound="-1" eType="ecore:EByte"/>
        <feature xsi:type="ecore:EAttribute" name="RuleActionClause"/>
        <feature xsi:type="ecore:EAttribute" name="RuleSelectClause"/>
    </feature>

    <feature xsi:type="ecore:EClass" name="MsgBuffer">
        <feature xsi:type="ecore:EAttribute" name="AppMsgBuffer" lowerBound="1" upperBound="-1" eType="ecore:EReference"/>
    </feature>

</package>
</.ecore>
```

201
<eClassifiers xsi:type="ecore:EClass" name="ExecutionTime1">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name" value="ExecutionTime1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="DefaultValue" value="2000" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MinValue" value="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MaxValue" value="10000" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="BufferLength">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name" value="BufferLength" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="DefaultValue" value="20" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MinValue" value="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MaxValue" value="30" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="SendingQueueLength">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name" value="SendingQueueLength" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="DefaultValue" value="10" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MinValue" value="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MaxValue" value="20" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Threshold">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name" value="Threshold" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="DefaultValue" value="50" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MinValue" value="40" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="MaxValue" value="100" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EDouble"/>
</eClassifiers>

Listing B.2 A XML file to describe the example PIM
Appendix C  Bibliography


91. IETF. *IPv6 over Low power WPAN (6lowpan)*. 2007 [cited 2014; Available from: https://datatracker.ietf.org/wg/6lowpan/.


Erklärung


Kassel, im Juni 2015

MSc. Nguyen Xuan Thang