ABSTRACT

LIM, JUN BUM. RaPTEX: Rapid Prototyping Tool for Embedded Communication Systems. (Under the direction of Assistant Professor Mihail L. Sichitiu).

Advances in microprocessors, memory, and radio technology have enabled the emergence of embedded systems that rely on communication systems to exchange information and coordinate their activity in spatially distributed applications. Developing embedded communication systems that are efficient and reliable, is a challenge due to the trade-offs imposed by the conflicts between application requirements and hardware constraints. In this thesis, we present RaPTEX, an integrated development environment (IDE) for embedded communication systems. RaPTEX consists of three major subsystems: a graphical module to facilitate component composition, code generation with access to component-level parameters, and a performance estimation framework for allowing system designers to explore what-if scenarios and clearly expose the trade-offs of their choices. We also present two case studies of developing wireless sensor network applications using RaPTEX.

RaPTEX: Rapid Prototyping Tool for Embedded Communication Systems

by

Jun Bum Lim

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, NC

2007

Approved By:

Dr. Injong Rhee	Dr. Laurie Williams

Dr. Mihail L. Sichitiu
Chair of Advisory Committee

Dedication

I dedicate this thesis to
my Parents,
my Wife,
and my Son.

Biography

Jun Bum Lim was born in Seoul, Korea on February 6, 1975. He received his Bachelors degree in Computer Science from Dankook University, Korea, in 2000. After his undergraduate studies, he worked for Samsung SDS and SSangyoung Information & Communications Corp. as a software engineer in Korea. In fall 2005, he enrolled in North Carolina State University as a graduate student to pursue Masters of Science in Computer Science. His interest lies in Wireless Sensor Networks and their applications.

Acknowledgements

First of all, I would like to thank my advisor, Dr. Mihail L. Sichitiu for his constant support and encouragement during my entire graduate studies at NCSU. His guidance was significant in the successful completion of this thesis. I would like to thank Dr. Injong Rhee and Dr. Laurie Williams for being on my thesis committee. I am also grateful to all my professors for making my graduate studies a great learning experience.

I would also like to thank Beakcheol Jang and Su Young Youn whose steadfast support of this project was greatly needed and deeply appreciated. This thesis would be incomplete without the support from Beakcheol Jang, and without Su Young's analytical models.

I am forever indebted to my parents, brother, and sisters for their understanding, endless patience, and encouragement when it was most required. I am also grateful to my father-in-law and mother-in-law for their support and understanding.

Finally, special thanks to my wife. She has patiently allowed me to study at the expense of household chores, holidays and countless other little things, all of which only love could endure. I dedicate this thesis to my parents, my wife, and my son, Ryan.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are a promising sensing technology for large geographical areas. A large number of inexpensive, small sensors are scattered in the sensing area. Each sensor node is capable of taking samples from its sensors, send them to its neighbors, and forward data from its neighbors toward a sink that collects the data and send it to the end user(s) via Internet or satellite. Fig.1.1 illustrates the general sensor network deployment [1].

Sensor networks can significantly improve the quality of the collected data and correspondingly the resulting information. Sensor networks can help to collect high fidelity information and enable sensing of hard-to-obtain information from the physical world. Through WSNs, users can control information that is precisely localized in time and/or space, according to the user's needs or demands.

Although WSNs enable the dream of ubiquitous computing, they still face many restrictions. Sensor nodes primarily rely on batteries and resource limited sensing devices, require an energy and resource efficient process. Sensor networks are deployed in an ad hoc fashion in remote, unattended, hostile environments, which require distributed and self-organizing algorithm. While many research groups focus their efforts on WSNs, the unique characteristics of WSNs make it difficult to productively optimize WSN systems.

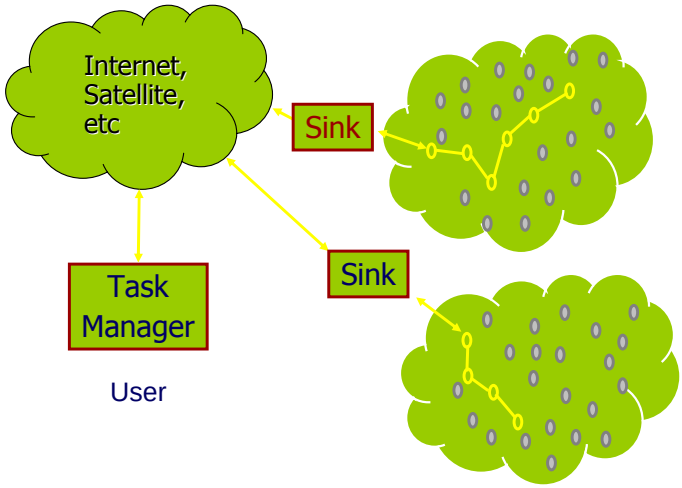

Figure 1.1: Sensor nodes scattered in a sensor field

1.2 Motivation

One of the biggest difficulties of developing WSN applications is the design of efficient communication protocols given the application requirements and the constrained resources of the sensor nodes. Although many WSN communication protocols [2–12] have been designed for each network layer of WSNs, only a few network specialists can build a protocol stack by assembling and customizing the parameters of the proposed protocols.

It is non-trivial and time-consuming task for application developers to find the optimal composition of communication protocols that meets the application requirements and WSN platform constraints. The problem is even more complicated for non-specialists that are not familiar with the available communication protocols.

1.3 Goal

The main goal of the work presented in this thesis is to bridge the gap between the need for WSN communication systems and the difficulty in their design. To achieve this goal, we provide RaPTEX, an Integrated Development Environment (IDE) with code generation for rapid prototyping featuring a performance estimation methodology that allows for immediate feedback on the impact of the design decision.

1.4 Contributions

The contribution of this thesis is the development of RaPTEX, which provides:

- a block composition Graphic User Interface (GUI) to facilitate development; the tool provides a library of components; users assemble the communication system from these components by simply dragging, dropping and wiring them;
- an automatic code generation with a component-level parameter passing scheme to allow users to tune the parameters of each protocol.
- tool that allows users to quickly estimate the performance of the resulting networking stack.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides TinyOS and WSN applications background and a summary of the related work. Chapter 3 describes the proposed IDE in detail. In Chapter 4, we present the generation sensor application development process and two case-studies: a periodic data gathering WSN application and an event-driven WSN application. Chapter 6 concludes the thesis.

Chapter 2

Background and Related work

In this chapter, we provide an overview of TinyOS, which is the underlying system platform for current version of RaPTEX. In addition, we classify sensor applications and discuss their design considerations. We conclude the chapter by presenting an overview of the related work.

2.1 TinyOS

TinyOS is by far the most popular operating system (OS) designed for wireless sensor networks. As an open source OS, TinyOS has been successfully ported to numerous hardware platforms, it is used by over 500 research groups and companies, and over 10,000 copies have been downloaded [13]. The design of TinyOS was driven by several challenges specific to WSNs, including low-power operation, concurrent operation, robustness, and flexibility [14, 15].

For low-power, robust, concurrent operation, TinyOS adopts an event-driven model that allows a sensor node to simultaneously execute several operations while only using limited resources. In this model, program execution is triggered by hardware events such as a fired timer, a sensor with new data, or a packet arrival at a communication device. Instead of using blocking or spin loops, operations are split-phase: operation request and completion are separate functions. For example, to send a packet, a component

may call the *send* command to start the transmission; a signal is triggered by the communication device when packet transmission has been completed. Although this approach increases the complexity of programming logic by mixing the sequence of operations, it is a good strategy for low power concurrent operations, because concurrent operations can be serviced on a single stack with very limited RAM. Also, if there are no tasks to be executed, the system can put the system to sleep until the next interrupt.

The other choice of TinyOS is the component-based architecture (in some ways similar to a object-oriented architecture). The modularity of the component-based approach allows application flexibility with respect to hardware diversity and application requirements. Even the basic system modules of TinyOS that implement the basic functionality such as basic I/O, timers, network stacks are components. Each application hierarchy consists of a set of system components and application-specific components that are written by the application developer, each component being interconnected through commands and events. Commands initiate downward operations that flow from higher level components to lower level ones. Events from lower level components are captured and signaled upward to event handlers in higher level components.

TinyOS is implemented in nesC [16], an extension of the C language designed to capture the structuring concepts and execution model of TinyOS. Applications are built by composing, and configuring nesC components.

2.2 Wireless Sensor Network Applications

Although applications of sensor networks are diverse ranging and the design of sensor networks is application driven, the existing applications can be broadly classified into two categories: continuous monitoring and event-driven [17, 18].

Continuous monitoring applications of sensor networks collect sensor data periodically and report the reading to the sink via a multi-hop routing protocol. Generally, this type of applications uses stationary networks that monitor the target area [19, 20]. For example, the mote-based tiered sensor network on Great Duck Island [19] monitors the environment and behavior of nesting seabirds. In this experiment, 32 motes were deployed in the area of interest, and transmitted their data to the gateways (CerfCubes) that were responsible for forwarding the data to the remote base station that provided

WAN connectivity and data logging. An important goal in this class of application is to maximize the life time of the network in an unattended environment; often, longer delays can be tolerated.

On the other hand, event-driven applications detect and/or track the occurrence of interesting events such as movement of an object or a certain degree of change for some sensor reading and report the occurrence of the event to the base station. There have been several deployments of applications in this category [21–23]. For example, the goal of TinyOS-based surveillance system [23] is to detect and track the positions of moving vehicles in an energy-efficient and stealthy manner. The authors describes the trade-offs between energy efficiency and system performance by adjusting the sensitivity of the system. Generally, in this type of application, reliability and delay are significant goals often as important as minimizing energy consumption.

Although applications in sensor network have specific goals, most applications share common characteristics. Maximizing network lifetime is a common design objective because in most case, the sensor nodes operate on limited battery power. Finally, due to limited resources they often use simple, distributed algorithms.

2.3 Related Work

There have been several projects that aim to facilitate development of wireless sensor network (WSN) applications. Some of them [24–26] provide an environment for developing WSN applications. Others [27–31] enable a comparison of possible solutions through simulation or emulation. Others [32] focus on developing WSN applications by providing high level language, libraries, and compilers.

TinyDT [24], Viptos [25] and GRATIS [26] are design environments for WSN applications based on TinyOS [14]. TinyDT is a TinyOS plug-in for the Eclipse platform that implements an Integrated Development Environment (IDE). It focuses on providing editing functionality such as syntax validation and highlighting, code navigation, automatic build support and static wiring graph analysis. Although these IDE facilities help, the graphical block composition of RaPTEX is much easier to use by non-specialists, as in RaPTEX the applications can be built by wiring preexisting nesC components.

Both Viptos [25] and GRATIS [26] feature graphical block composition interfaces

that provide a clear overview of the TinyOS application structure and make it possible for non-specialists to easily develop WSN applications by simply choosing preexisting blocks. However, they both limit the flexibility of the configuring the components. Designers may want to change the parameters of each component to efficiently customize their code. For example, a designer may choose SCP-MAC [5] as the MAC protocol and may want to customize parameters such as the SYNC and channel polling interval that significantly affect its performance (essentially providing a trade-off between power efficiency and delay). Viptos and GRATIS do not allow such a configuration. In contrast, RaPTEX provides a component level parameter passing scheme, allowing the designers to customize these parameters to meet application-specific requirements.

Viptos provides support not only for code development but also a simulation environment. Viptos integrates the TinyOS simulator, TOSSIM [33], and the network simulator, VisualSense [34], through the Java Native Interface (JNI). Prowler [31] is a probabilistic WSN simulator developed in Matlab. It provides deterministic and probabilistic modes for simulating non-deterministic communications. SensorSim [27] is built on NS-2 [28] and provides sensor channel models and power models. It offers a hybrid simulation environment, in which real sensor nodes can participate in the simulation.

Simulation is one of the most popular methods for estimating the performance of network systems. Several network simulators that can be also used for wireless networks have been developed. NS-2 [28] is a discrete event simulator that provides support for simulating many network protocols. The UCB Daedelus and CMU Monarch projects added wireless extensions. OPNET [29] and OMNET++ [30] are also discrete event simulators providing support for wired and wireless communications. The former uses finite state machines (FSM) to define the protocols; while the latter defines component modules and interfaces with an object-oriented approach.

The primary advantage of network simulation is that they can predict the performance of network programs without expense in time and money required for a testbed. On the other hand, the main drawback of network simulation is its scalability with the number of nodes simulated. The performance estimation result of Viptos shows the simulation time linearly increases proportionally to number of nodes [25]. In contrast, the performance estimation environment of RaPTEX leverages theoretical results, and thus its running time is always small, constant and independent of the number of nodes.

SNACK [32] is a sensor network application construction kit built on nesC and TinyOS. The authors point out that the efficiency of WSN application is especially important due to the limited hardware resources of the sensor platforms, which in turn results in a high level of programming complexity (i.e., system-level code manipulation). To address this problem, SNACK provides a new component composition language, application-level service libraries and a new compiler. The component composition language of SNACK helps users to develop efficient WSN applications; however, the graphical block composition environment of RaPTEX is much easier to use, especially by non-specialists. The high-level libraries of SNACK make it possible for users to customize predefined parameters; however it requires a significant effort to build the libraries for each component, and the library interfaces are not intuitive. On the other hand, the parameter-configuration in RaPTEX does not require additional libraries and is more intuitive because it is done through a GUI.

Chapter 3

Design

As an IDE, RaPTEX has a block diagram-based GUI environment, a code generation with component level parameter passing mechanism, and a performance estimation framework, as shown in Fig.3.1. The details are discussed in following subsections. The current version of RaPTEX was designed to work with TinyOS components, but can be expanded to work with other component libraries. In this thesis, we focus on TinyOS 1.x and nesC 1.2 as underlying platform and language.

3.1 Block Diagram-based GUI IDE

Figure 3.2 shows the RaPTEX graphical user interface (GUI). The main window includes several panels (which can resized or detached if needed), each with a different functionality. The component library (a) stores all nesC components available in TinyOS. These components can be dragged in the diagram panel (b) where they can be connected to each other, grouped into compatible components, and ungrouped into the constituent components. Each component is shown in a different tab in the diagram panel. The configuration panel (c) allows the users to change the parameters of any of the component in the diagram panel to explore the influence of that parameters on the performance metrics displayed in panel (d). Finally, feedback messages to the user are displayed in panel (e).

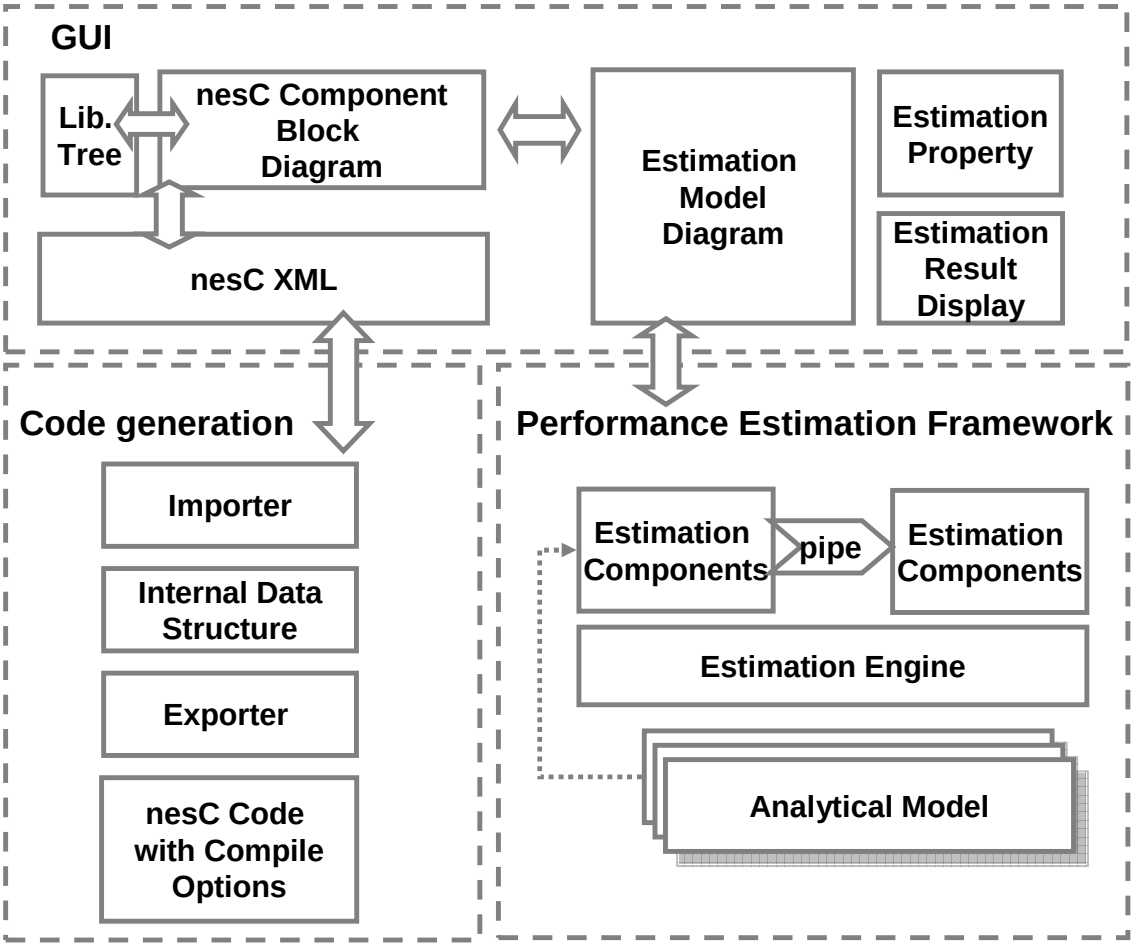

Figure 3.1: RaPTEX software architecture overview

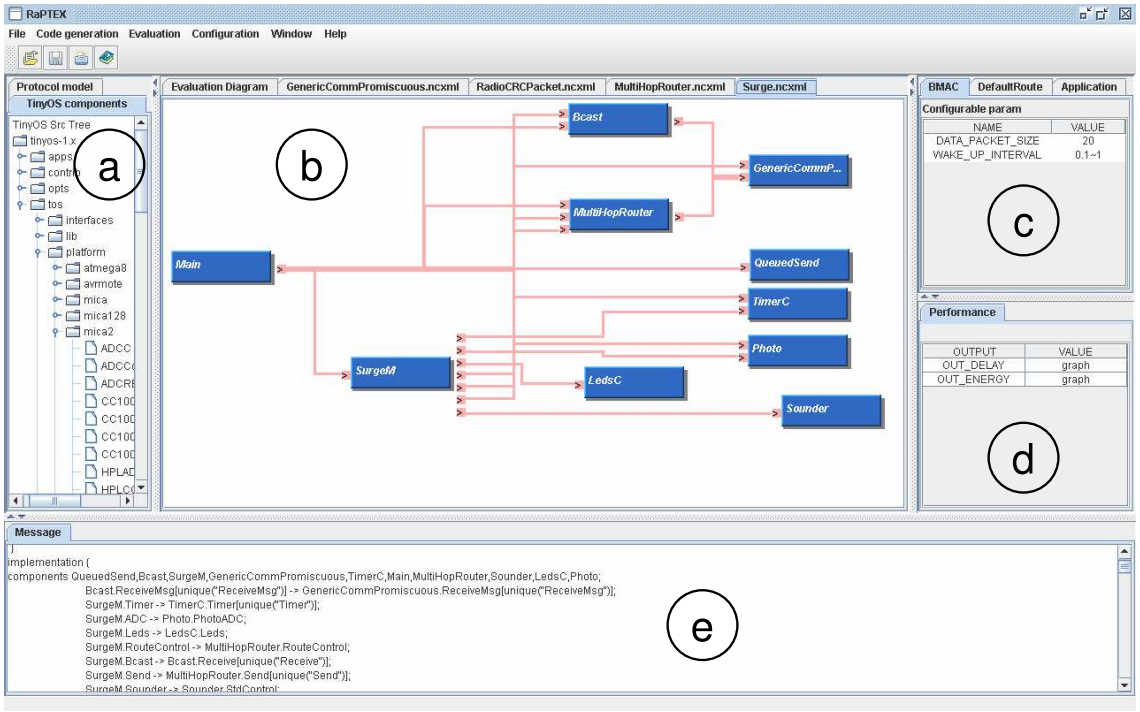

Figure 3.2: The RaPTEX user interface for Surge application.

RaPTEX's GUI allows the user to easily and efficiently handle code components by manipulating graphical elements. The goal is to improve the programming environment for sensor application and facilitate system design. The component-oriented organization in nesC enables a graphical interface that is much more intuitive than text base environments. The RaPTEX GUI allows the users to graphically display component wiring, a TinyOS component library tree with available choice of components, and compilation facilities for code generation.

The design of RaPTEX's GUI is based on Model-view-control (MVC) architecture [35] for flexibility. In RaPTEX, the nesC components are viewed as blocks, with the application as a top level component that contains interconnected sub-components. To represent the nesC components, we use the existing capability of nesC 1.2 of generating outline information of nesC components in XML format by using the -fnesc-dump= option of the compiler [36]. Viptos also uses this XML output feature; we improved the wiring information by removing redundant language transformation process. In Viptos, which works on top of Ptolemy II [37], the nesC file is converted to MoML [38], which is the internal representation scheme of Ptolemy II for interconnectable components. For this process, Viptos makes a two pass transformation: the nesC file is first transformed into nesC XML, which is then transformed into MoML. However, as pointed out in [25], certain type of nesC configuration cannot be expressed in Viptos when translating from nesC to MoML because in Viptos multiple connections between components form a relation group in which the relations involved are indistinguishable from each other and the connection between relations are directionless. By using the nesC XML output directly as an underlying data representation scheme, we eliminate information loss during translation and fully captures nesC's semantics. Figure 3.3 shows nesC XML which corresponds with original nesC code in Fig. 3.4 and block diagram in Fig 3.2.(b).

The components available in RaPTEX are organized as a tree constructed from the XML resulting from the nesC translation; this library tree mirrors the TinyOS source directory structure. From the component library tree, users may choose components for their application by dragging and dropping them into the diagram panel and make connection by clicking on the interfaces of each component. The resulting diagram is saved as nesC XML format and the code generation module uses this information to generate nesC code (details in Section 3.2).

```
<nesc xmlns="http://www.tinyos.net/nesC">
  <components>
    <component qname="Surge" loc="39:/opt/tinyos-1.x/apps/Surge/Surge.nc">
      <configuration/>
      <wiring>
        <wire loc="51:/opt/tinyos-1.x/apps/Surge/Surge.nc">
          <from><interface-ref name="StdControl" ref="0xb7de6c18"/></from>
          <to><interface-ref name="Control" ref="0xb7d20c58"/></to>
        </wire>
        ...
        <wire loc="62:/opt/tinyos-1.x/apps/Surge/Surge.nc">
          <from><interface-ref name="RouteControl" ref="0xb7db27f8"/></from>
          <to><interface-ref name="RouteControl" ref="0xb7aab8a0"/></to>
        </wire>
      </wiring>
    </component>
  </components>

  <interfaces>
    <interface provided="0" name="StdControl" ref="0xb7de6c18" ..>
      ...
    </interface>
      ...
    <interface provided="0" name="RouteControl" ref="0xb7db27f8" ...>
      <component-ref qname="SurgeM"/>
      ...
    </interface>

  </interfaces>

    ...

</nesc>
```

Figure 3.3: nesC XML for Surge.nc.

```
configuration Surge {
 }
implementation { components
    QueuedSend,Bcast,SurgeM,GenericCommPromiscuous,TimerC,
    Main,MultiHopRouter,Sounder,LedsC,Photo;

    Bcast.ReceiveMsg[unique("ReceiveMsg")]
        -> GenericCommPromiscuous.ReceiveMsg[unique("ReceiveMsg")];

    SurgeM.Timer -> TimerC.Timer[unique("Timer")];
    SurgeM.ADC -> Photo.PhotoADC;
    SurgeM.Leds -> LedsC.Leds;
    SurgeM.RouteControl -> MultiHopRouter.RouteControl;
    SurgeM.Bcast -> Bcast.Receive[unique("Receive")];
    SurgeM.Send -> MultiHopRouter.Send[unique("Send")];
    SurgeM.Sounder -> Sounder.StdControl;
    Main.StdControl -> SurgeM.StdControl;
    Main.StdControl -> Photo.StdControl;
    Main.StdControl -> MultiHopRouter.StdControl;
    Main.StdControl -> GenericCommPromiscuous.Control;
    Main.StdControl -> Bcast.StdControl;
    Main.StdControl -> TimerC.StdControl;
    Main.StdControl -> QueuedSend.StdControl;
    MultiHopRouter.ReceiveMsg[unique("ReceiveMsg")]
        -> GenericCommPromiscuous.ReceiveMsg[unique("ReceiveMsg")];

 }
```

Figure 3.4: Generated nesC source code for Surge.nc.

Some of the choices made during design phase do not translate directly in nesC glue code, but rather in compile-time option. For example, for MAC protocols, there are many choices with different energy consumption, delay, and throughput. However, many sensor MAC protocols are wired through the same named RadioCRCPacket component with different implementations to provide compatibility to the Active Message interface of the network layer [15]. If users want to explore trade-offs among different MAC protocols, the RadioCRCPacket of the considered MAC protocol should be dragged onto diagram panel. RaPTEX will then change the library path during the compilation without any change of the source code.

3.2 Automatic Code Generation with Parameter Passing Mechanism

Automatic code generation has the potential to revolutionize application development by allowing software system to easily generate error-proof code, rather than hand-crafting each application [39]. RaPTEX provides a nesC code generator with a model-driven design. In addition to code generation, we introduce a component level parameter passing scheme.

The model-driven architecture of RaPTEX's code generator has three submodules: an importer, an internal object model, and an exporter. The importer reads the user's configuration information and constructs the internal object model, which is an object-oriented design implementation of the nesC structure and the core of the code generator architecture. Each element of the internal object model corresponds with a part of a nesC component model. The NCDiagram models nesC's top-level component that defines and connects the nesC subcomponents. The NCComponent corresponds to nesC subcomponents, and their interfaces are represented with the NCLinkableComponent, which belongs to a NCComponent. By connecting two NCLinkableComponents with an NCLink, NCComponents can be linked to other components through their NCLinkableComponent object. To generate the code, the exporter extracts the relevant information from the internal object model, and generates the code corresponding to the model. Figure. 3.4 shows nesC code for Surge generated by the RaPTEX code generator.

RaPTEX also provides a component-level parameter passing mechanism. The parameters in this context are public variables of a component that are tunable from the GUI. By default, nesC does not explicitly support configuration-level parameters. Although nesC supports a syntax for "parameterized interfaces", this is used to support multiple instances of the same interface for a single component [16]. Therefore, parameters are often set by *#define* or *enums* in header files. In other cases, a component such as BMAC [4] provides predefined primitives to allow customization from outside the component. However, all these schemes are source level approaches that have a steep learning curve and preclude wide usage. To overcome this limitation, SNACK [32] generates nesC code with configuration-level parameters through high level services and a proprietary language. The high level services function like a template file of nesC containing marking codes, and the language is used to glue the services and set parameters. When the SNACK compiler processes the language, it replaces the marks with user's configurations and generates nesC code by gluing the service files and copying the service files with replaced strings. Within the boundary of high level services, SNACK facilitates application design, but the process of developing configurable services is not trivial. Moreover, SNACK requires learning yet a new language and compiler-specific functionality.

During the design of RaPTEX, we defined the requirements of a parameter passing mechanism:

- The first and most important requirement is not to change the nesC code of TinyOS, thus ensuring that RaPTEX can use future versions of TinyOS with no modification.
- Second, for efficient and compact execution in embedded systems, parameters should be passed at compile time rather than at execution time.
- Finally, parameters should be explicitly exposed to users through the GUI.

To meet these requirements, RaPTEX uses several features of the nesC compiler. To define parameters in a component, RaPTEX first use nesC's annotation, which allows simple language extensions without introducing new keywords and burdening the syntax. Parameters are passed to a component through regular compile option (-D option). The RaPTEX GUI interprets defined parameters of a component and exposes them to the users (Fig. 3.2(c)). If configurable values are already defined by *#define* in the source code, parameters passed through the -D option override the values in the source code. However,

if a component cannot work with -D option, we provide a wrapper component based on nesC design patterns [40] such as Decorator and Facade with *#ifdef* statement. A wrapper is just a regular nesC component which is responsible for defining configurable parameters, discriminating parameters passed through -D option, and manipulating behaviors of an encapsulated component by changing compositions or calling predefined primitives at compile time. From the nonspecialists' perspective, all the detail of the implementation of the wrapper and the parameter passing processes are hidden by RaPTEX. Therefore, they can easily explore trade-offs by changing the parameters through the GUI.

3.3 Performance Estimation Framework

Many WSN communication protocols have been designed for each network layer of WSNs [2–12], and they can be composed into many different networking stacks. Moreover, the parameter-setting facility in RaPTEX dramatically increases the number of possible instances of networking stacks. Several simulation-based estimation frameworks for WSN [25–27,31] have been developed; however they are time-consuming, especially for large networks; the direct consequence is that the designer is less inclined to explore a large number of combinations of protocols and parameters to optimize the performance of his or her application. RaPTEX provides an object-oriented performance estimation framework based on theoretical analysis, in which designers can quickly and easily estimate the performance of their current protocol stack.

As shown in Fig. 3.1, the RaPTEX estimation framework consists of four basic software elements: the engine, the component, the pipe, and the analytical model:

- The engine acts as a controller that coordinates the estimation process.
- The component provides a template, as a super class for an analytical model for each protocol, and it is implemented as a java class. For example, to implement BMAC's analytical model the estimation component for BMAC should extend the base class (EvalComp) and a network system developer (see Section 4.1 for the different types of users of RaPTEX) overrides the methods in EvalComp. Then, the BMAC component can be used in RaPTEX's performance estimation frame by being connected with other models by the engine.

- The input and output pipes are defined for each component and connect upper layer components to lower layer components; they are used to pass the result of calculations from one layer to the other.

- The analytical model for each protocol provides the mathematical formula that captures performance metrics such as power, delay, etc.; the model is implemented in the component corresponding to the protocol implementation. In this thesis we do not propose any new analytical models, but rather use existing models [41] either developed specifically for RaPTEX, or published with the protocols themselves (e.g., the power consumption and delay models for B-MAC and SCP are available in [4] and [5], respectively).

The performance estimation framework in RaPTEX has a separate GUI. The engine is represented as an estimation diagram very similar to the nesC diagram, but not identical. While the nesC diagram represents the relation between the nesC components, the estimation diagram represents the relations between the analytical model components. When developers import a nesC application, RaPTEX uses the wired nesC diagram of the application to find the analytical model components that correspond to each nesC component, and automatically builds the estimation diagram that reflects the network stack of the imported application. Developers can set the values of the parameters for each protocol. The engine starts the estimation from a user-defined start component and finishes at an (again user-defined) end point component; in the process, each component passes its input and output values through the pipes connecting them. The final value of interest (e.g., power, delay, throughput, etc.) is saved in output pipe of the end point component. If for any parameter the user specifies a range, the result is displayed as a graph, as shown in Fig. 4.4.

Chapter 4

Case study

In this section, we outline the general sensor application development process using RaPTEX, and present two case studies of developing WSN applications: a continuous monitoring and an event-driven application.

4.1 The Development Process

We define three types of actors who are involved in a sensor application development using RaPTEX, and, accordingly, divide the development process into three phases. Fig. 4.1 illustrates the general development process in the context of actors and development phases.

- **Application developers** that have basic knowledge about developing sensor applications, but do not understand network protocols in detail. They build the application by composing and configuring existing components.
- **Network system developers** that understand the existing network protocols, the role of the parameters and analytical models; they play a role in implementing nesC wrapper components for parameter handling and estimation components in the RaPTEX's performance estimation framework.
- **Protocol designers** that devise new protocols and provide analytical models for

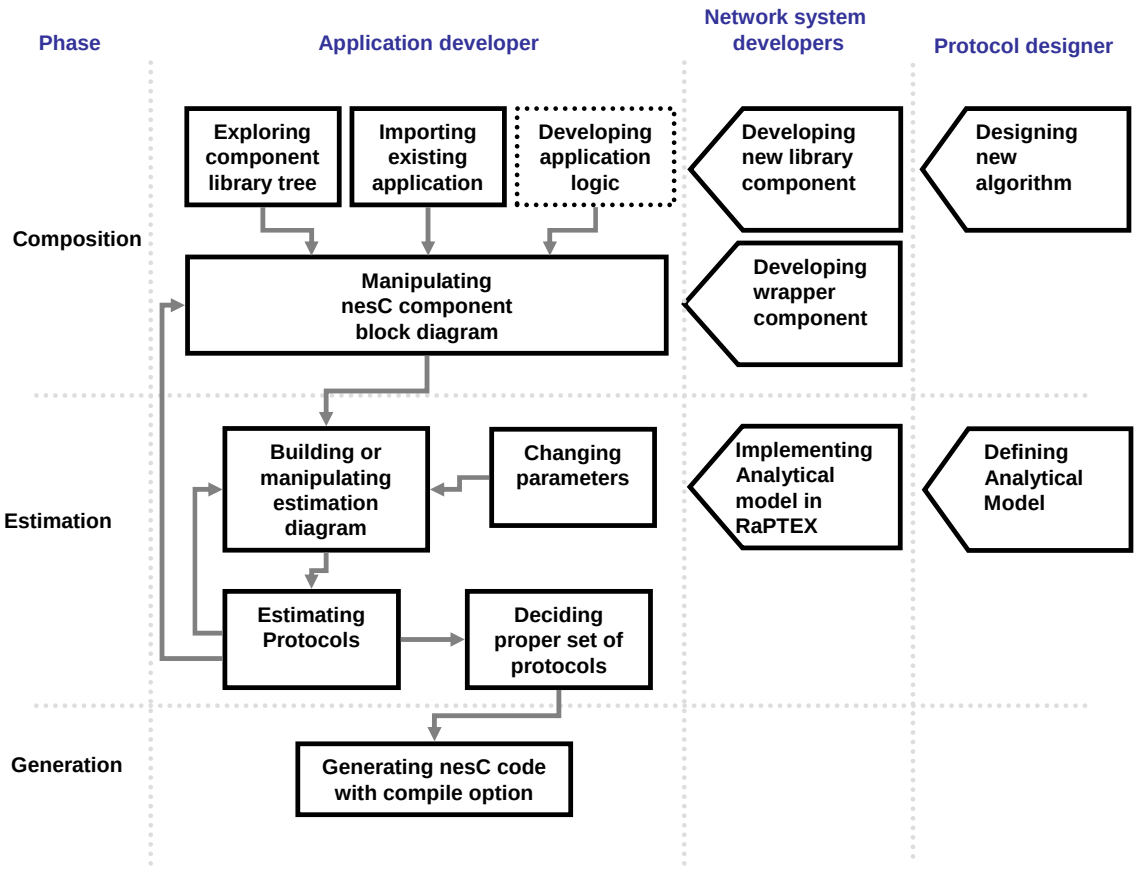

Figure 4.1: Application development process and roles involved in using RaPTEX.

each protocols.

Each box in Fig. 4.1 represents actions taken by either actors or RaPTEX modules. The first column shows the three phases of the development process: composition, estimation, and execution. The remaining three columns correspond to the actions of the corresponding actor.

RaPTEX provides tools to application developers with support from network system developers and protocol designers. A network system developer is responsible for implementing new library components, which can be imported into RaPTEX. In order to provide application developers with configurable nesC components, a network system developer needs to develop wrapper classes in RaPTEX. For performance estimation, a protocol designer should provide analytical models for their protocols (most papers provide models for key performance metrics for the proposed protocols); the network system developers implement the model by extending base classes of the RaPTEX performance estimation framework. The following two case studies are primarily focused on the application developers at each phase of the development process.

4.2 A Continuous Monitoring Application

In this section, we show the development of the simple periodic temperature monitoring application, Surge [16], with emphasis on optimizing the performance of the networking stack. The application samples the temperature every 600 seconds and uses an ad-hoc multi-hop routing protocol to deliver samples to the sink. While there can be many optimisation criteria that can be considered (e.g., lifetime, delay, fairness, reliability, etc.), for this application we will focus on maximizing the network lifetime.

The three phases of the development process are as follows:

First, the developer goes through the composition phase: RaPTEX provides the TinyOS component library tree (Fig.3.2 (a)) and the diagram panel (Fig.3.2.(b)). In this case study, the developer start from empty diagram and builds the application by placing TinyOS system components such as Main, TimerC, and LedsC from the component library tree and assembling these components with each other and with the application logic component (SurgeM) on the diagram panel. In this example, the developer will use the timer component in the application logic component because the monitoring application

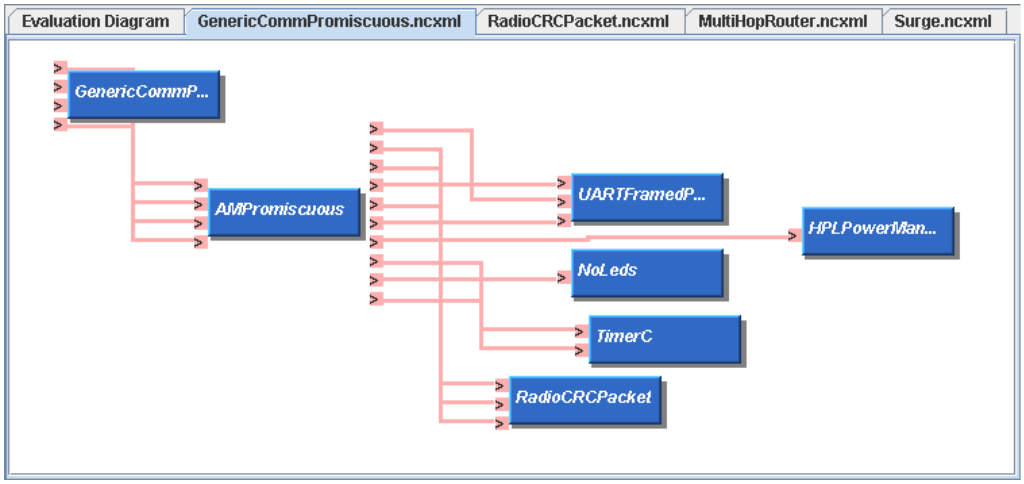

Figure 4.2: Sub-components of the GenericCommPromiscuous component.

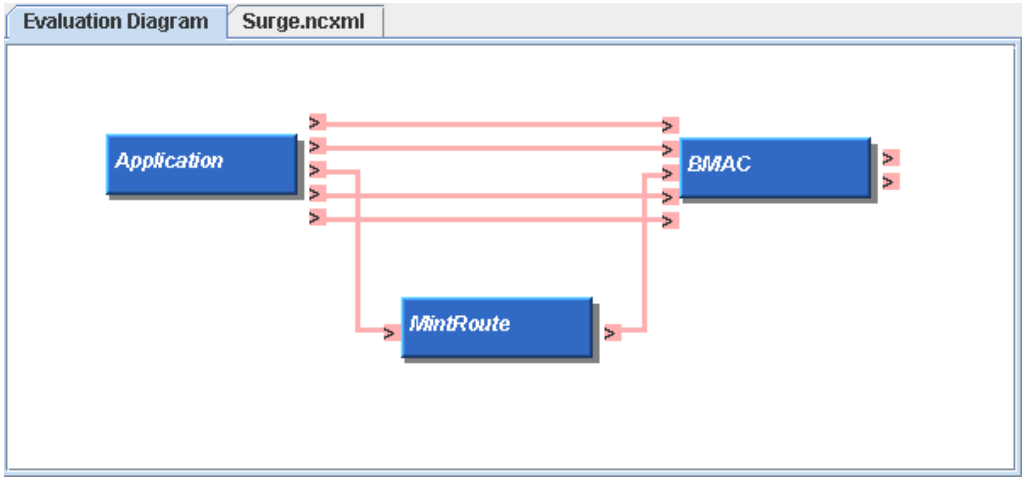

Figure 4.3: Default estimation diagram for Surge.

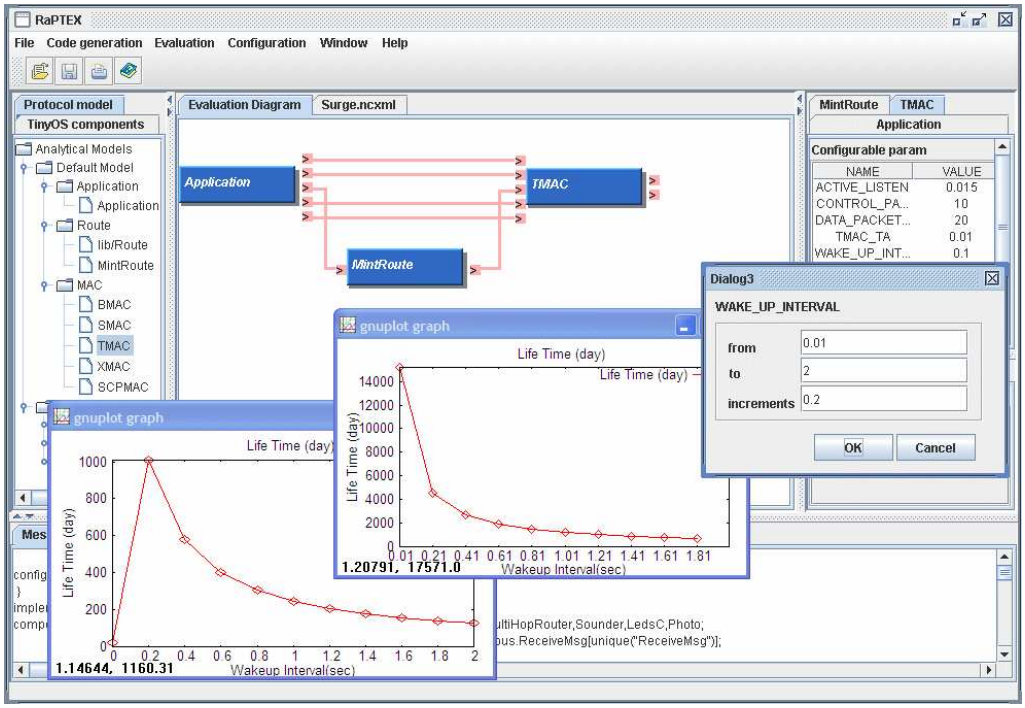

Figure 4.4: Performance estimation for BMAC.

periodically initiates the sensor reading and packet transmission when the timer fires. In the application logic, a timer device is manipulated through the Timer interface, but the actual linkage between the interface and real hardware is deferred until wiring is performed. By placing the TimerC component from the TinyOS library tree and wiring this component with SurgeM through the Timer interface, the application logic in SurgeM is wired with a timer device. Figure 3.2(b)) shows the completely wired component diagram for the Surge application.

Because a sensor application is composed of several components in hierarchical manner Due to the hierarchical structure of TinyOS applications, RaPTEX allows the developer to examine the content of any component by choosing *explore* on the pop-up menu of each component. Figure 4.2 shows the sub-components of GenericCommPromiscuous. While building nesC component diagram, RaPTEX automatically builds the estimation diagram that reflects the protocol stack used by nesC. Figure 4.3 shows the default estimation diagram of the Surge application, where the application developer choose B-MAC at the MAC layer.

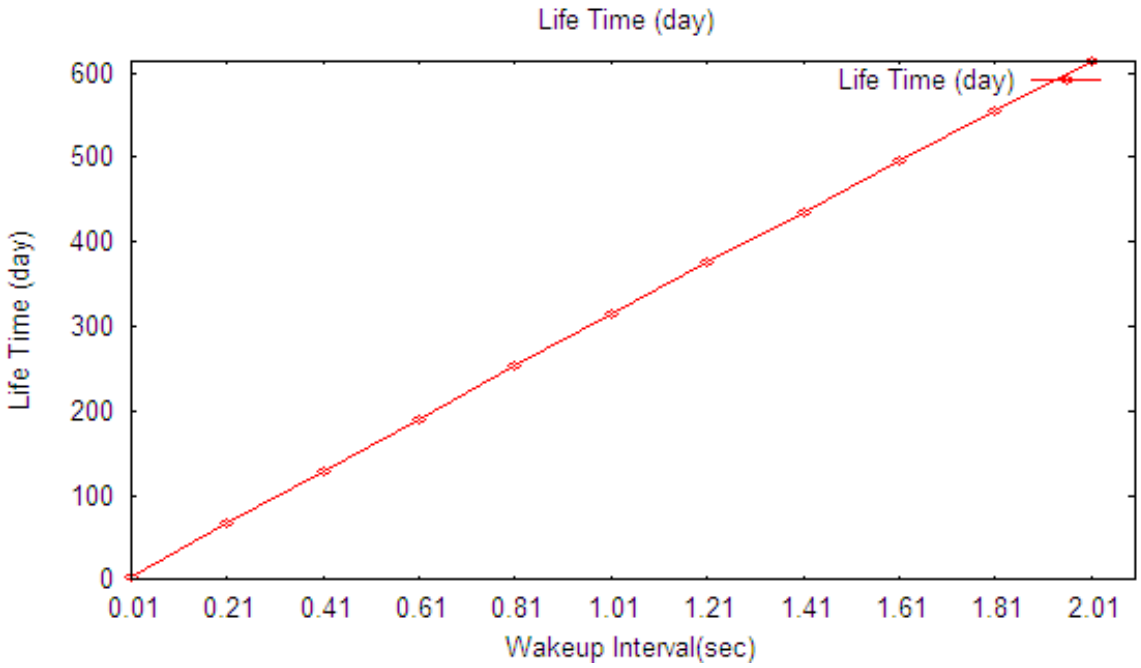

Figure 4.5: Lifetime as a function of wakeup interval for SMAC

At the second step, the application developer proceeds with the performance estimation phase, in which he or she estimates the resulting networking stack and changing its parameters. The developer can set the values of the parameters of each protocol in the property panel (Fig. 3.2.(c)). For example, the developer sets the range of wakeup interval of B-MAC from 0.01 to 2 seconds to see its effect on the lifetime of the resulting networking stack. Using the analytical model from [4], RaPTEX computes and displays the figure presenting the variation of the lifetime as the function of wakeup interval (e.g., Fig. 4.4). The developer may change B-MAC to other MAC protocols such as S-MAC, T-MAC and SCP-MAC, and obtain similar figures for these protocols.

From user's configuration, RaPTEX plots the result of performance estimation for each protocol: SMAC (Fig.4.5), TMAC (4.9), BMAC (4.7), and SCP-MAC (4.5), respectively. For comparison purposes, we plot all result from RaPTEX's estimation data in Fig. 4.10, which shows the lifetime as a function of wakeup interval for all MAC protocols. Because preamble-based MAC protocols such as BMAC send long preamble to wake up receivers, the preamble size increases with the wakeup interval As a result, the lifetime of a node decrease with an increase of the wakeup interval.

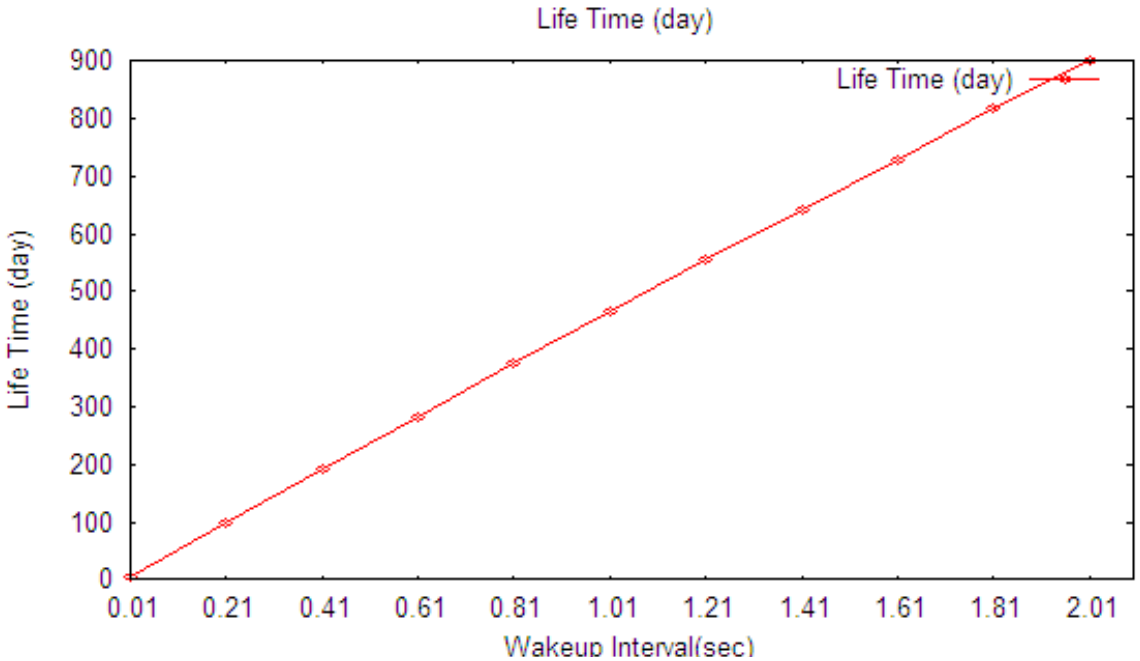

Figure 4.6: Lifetime as a function of wakeup interval for TMAC

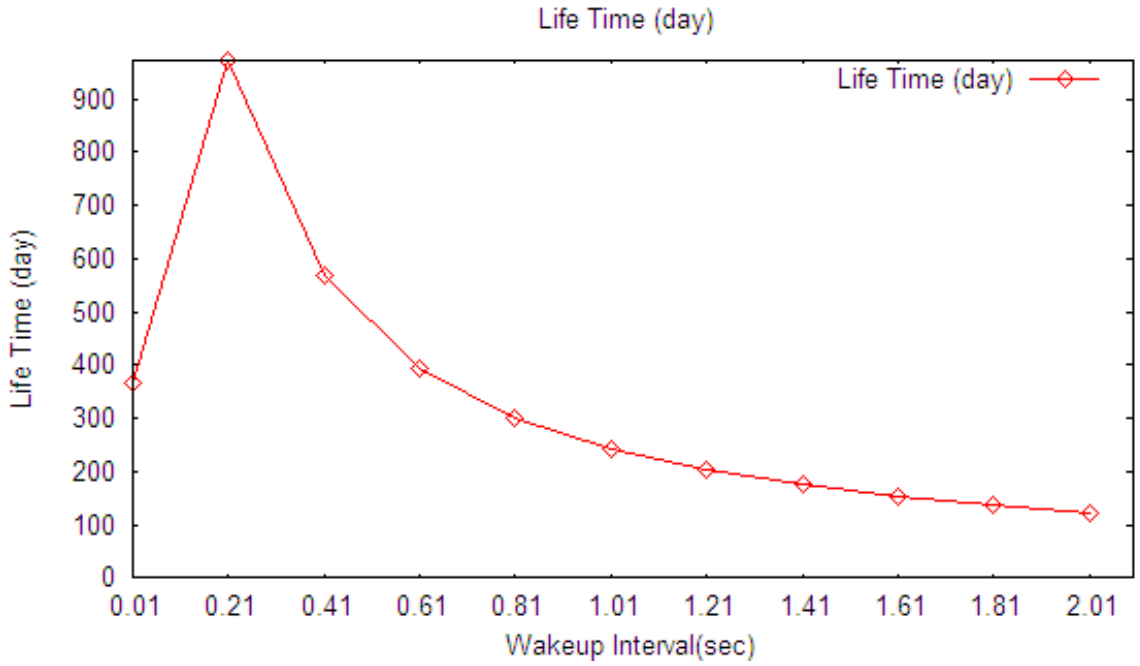

Figure 4.7: Lifetime as a function of wakeup interval for BMAC

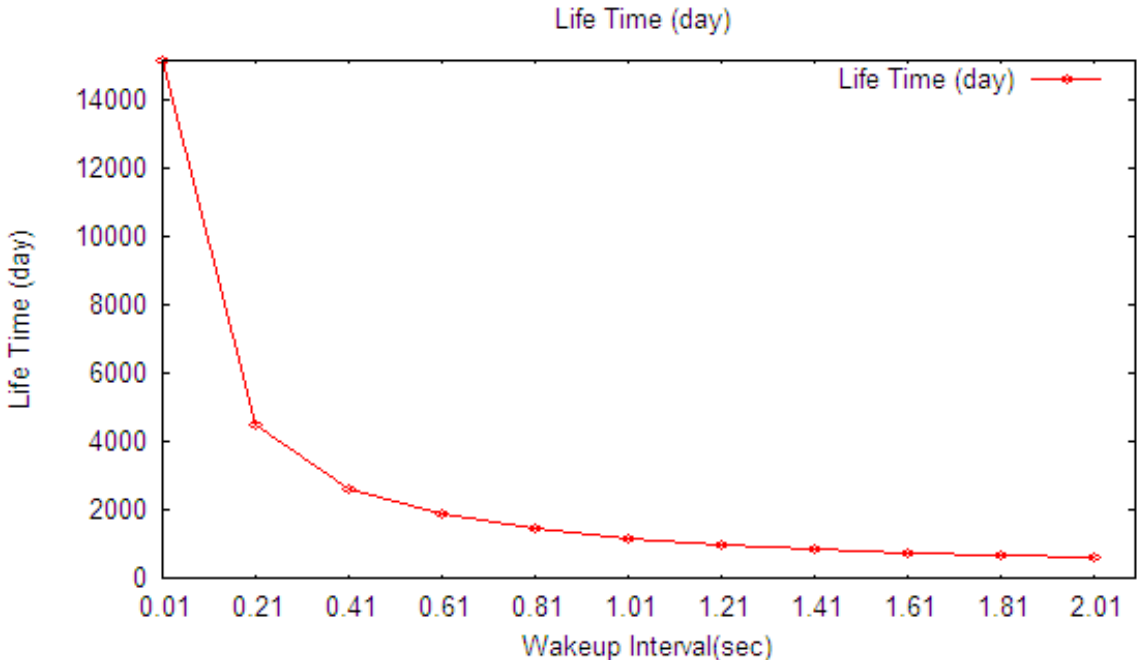

Figure 4.8: Lifetime as a function of wakeup interval for XMAC

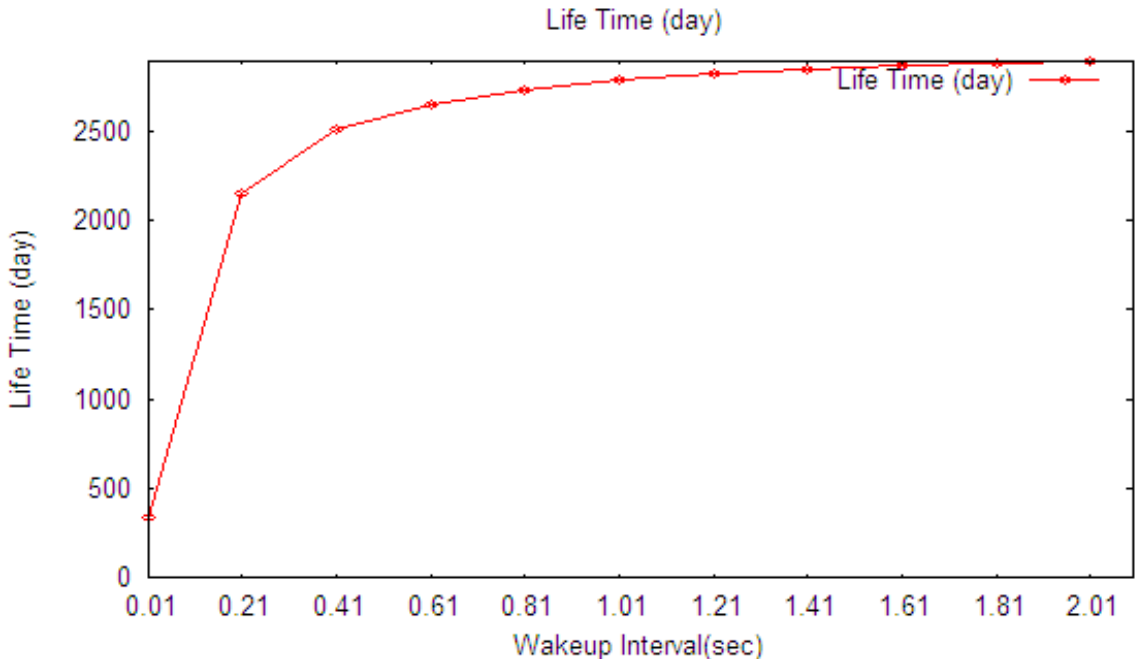

Figure 4.9: Lifetime as a function of wakeup interval for SCPMAC

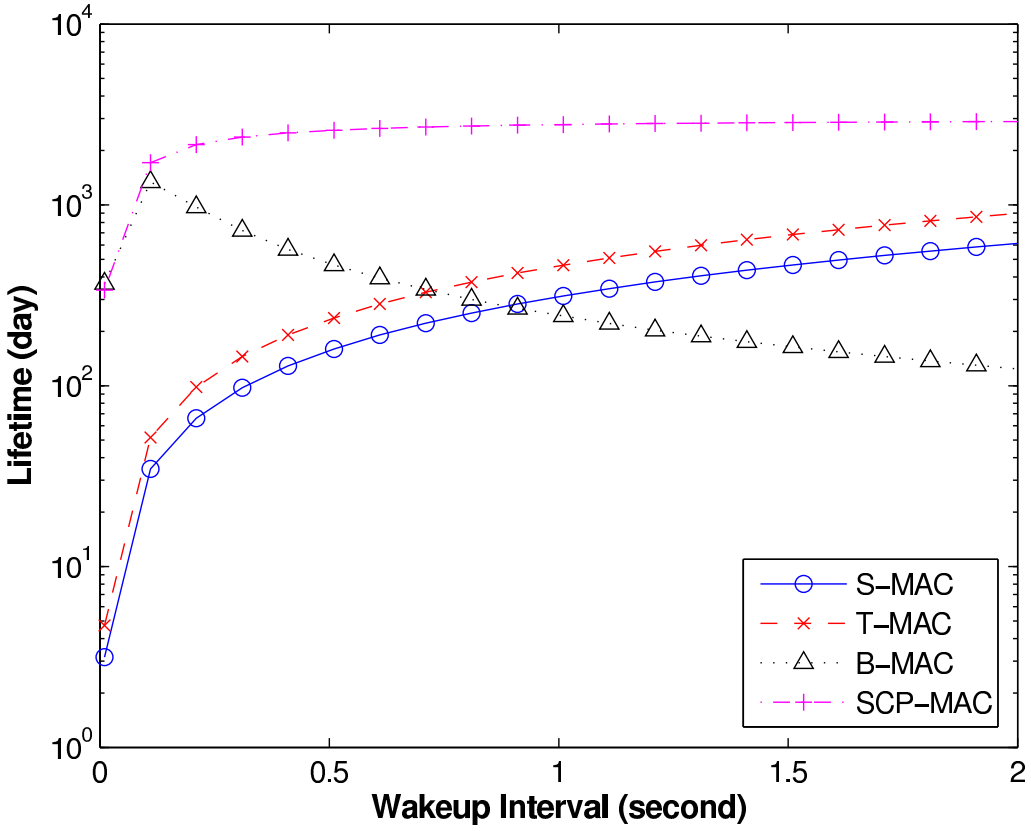

Figure 4.10: Lifetime as a function of the wakeup interval for various MAC protocols.

On the other hand, in synchronization-based MAC protocols such as SMAC, TMAC, and SCP-MAC neighboring nodes wake up and sleep at the same time. Therefore, nodes instead of sending long preambles they use periodic SYNC packets to maintain synchronization with their neighbors. As the wakeup interval increases, sensor nodes spend more time sleeping.

Using results from Fig. 4.10, the developer can explore trade-offs among different MAC protocols while considering the application requirements. For example, if the lifetime requirement is 2000 days, the developer may decide that SCP-MAC with a wakeup interval larger than 0.2 second is the optimal MAC protocol that meet his requirement based on Fig. 4.10. If the lifetime requirement is 100 days, the developer may choose anything among SCP-MAC, B-MAC, S-MAC with a wakeup interval longer than 0.4 second, or T-MAC with the wakeup interval longer than 0.3 second.

Finally, after deciding on the optimal composition of the networking stack, the developer saves the composition and the parameter settings (e.g., SCP-MAC with 0.2 second of wakeup interval). To generate code, the developer may choose *Run nesC* in main menu bar, then the nesC code is generated and can be saved for compilation.

4.3 An Event-driven Application

In this section, we the development process of an event-driven application, which, we assume, detects a fire and reports the event to a remote base station. Unlike in the case of a continuous monitoring application, the sensor nodes in event-driven application do not generate data packets until event of interest occurs. The application in this case study samples the temperature at the same rate as the continuous monitoring application in Section 4.2. If the sensor reading is higher than a threshold, the sensor sends the temperature (multi-hop) to the sink. We assume that this application requires a networking stack that maximizes the lifetime of WSN but also provides an upper-bound on the delay.

Depending on the definition of the event, the two classes of sensor networks share several implementation details: in many cases, a continuous monitoring behavior is necessary to detect an event, and other underlying behaviors such as multi-hop routing and hardware access are similar for both applications. Therefore, in the composition phase of this case study, the developer reuses and modify the continuous monitoring application

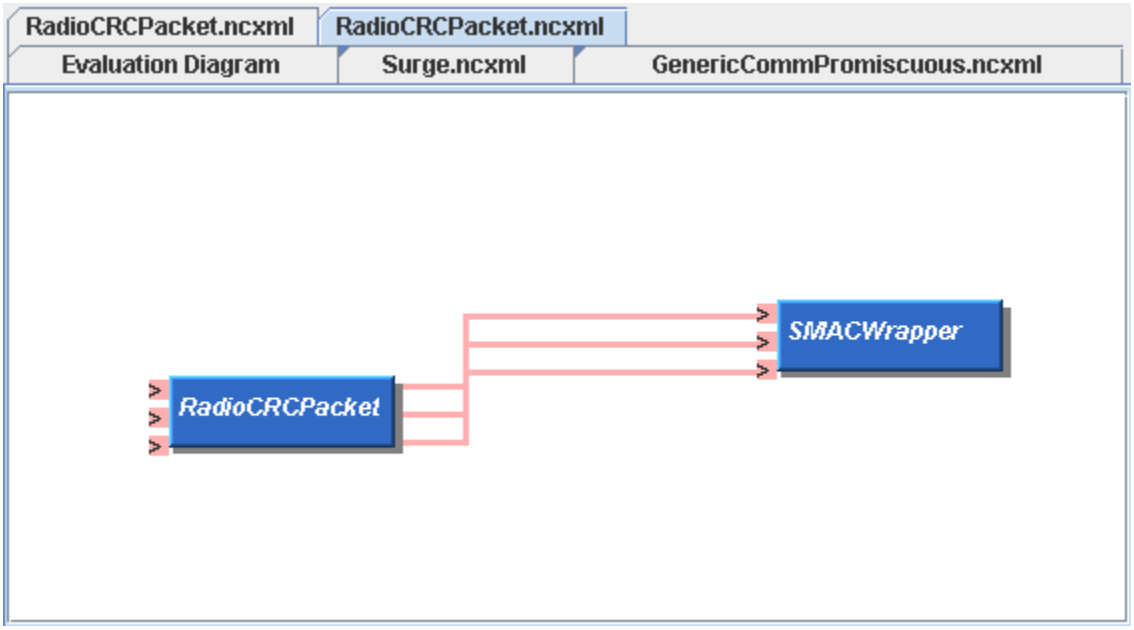

Figure 4.11: Inside of RadioCRCPacket for SMAC

in Section 4.2. The developer starts by importing the continuous monitoring application and change only the application logic component from SurgeM to SurgeMEvent that includes the event checking logic after reading the sensor. Fig.4.12 shows the scrap of SurgeMEvent.nc. To meet the different application requirements, the developer may want to change the protocols in the stack. To change protocol components, the developer can directly change wiring for *SendMsg* and *ReceiveMsg* interface to other protocols or change RadioCRCPacket in GenericCommPromiscuous (Fig. 4.2). For example, to change the MAC protocol from BMAC to SMAC, the developer simply places the RadioCRCPacket in SMAC directory over the BMAC's RadioCRCPacket. The change is also reflected in the estimation diagram. Fig.4.11 show inside of RadioCRCPacket that has wiring to SMACWrapper.

To support an event-driven application scenario, we extend the analytical models with three assumptions and one more parameter. We assume that:

(a) there is no in-networking process,

(b) events occur at a given event-rate

```
async event result_t ADC.dataReady(uint16_t data) {
  atomic {
    if (!gfSendBusy) {
      gfSendBusy  = TRUE;
      gSensorData = data;
      //To support event-driven style ..
      if(checkIfEvent(gSensorData)){
          post SendData();
      }
    }
  }
  return SUCCESS;
}
```

Figure 4.12: nesC source code of SurgeMEvent.nc

(c) every node is exposed to see the same rate of events.

The event-rate is the probability with which the event occurs during the fixed sample period. For example, if the event-rate is 0.5, then, on the average, one event occurs during two sample periods.

From user's configuration, RaPTEX plots the result of performance estimation for each protocol: SMAC (Fig.4.13), TMAC (4.17), BMAC (4.15), and SCP-MAC (4.13), respectively. For comparison purposes, we plot all result from RaPTEX's estimation data in Fig.s 4.18 and 4.19, which shows the lifetime of the network as a function of the event rate for two per-hop delay bounds. As the event-rate increases, the number of data packets during the (assumed fixed) sample period increase. As a result, the lifetime of the preamble base protocol such as BMAC decreases because the total number of packet to be sent increases. At very low event-rates and small wakeup intervals, the lifetime of BMAC is relatively high because in this case nodes rarely send data packets and the preamble size can be small.

On the other hand, at short wakeup intervals, the lifetime of SMAC and TMAC is relatively low because the sleep period reduces and both protocols send SYNC packets very often. However, the lifetime does not change as the event-rate increase because energy for sending the data packet is compensated by the decrease in the idle listening time.

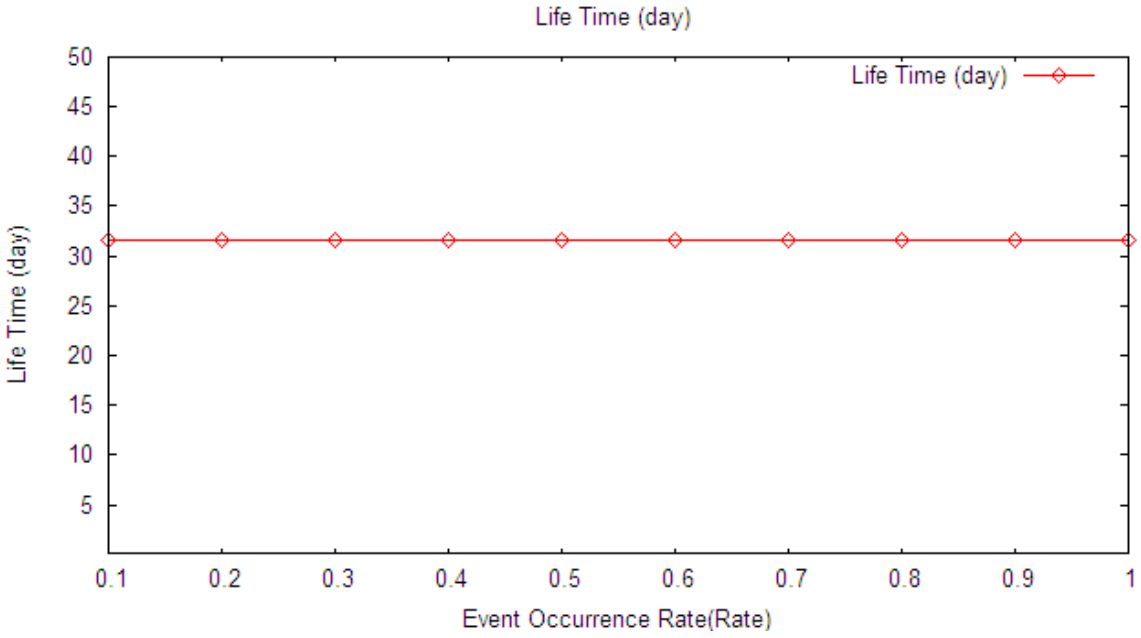

Figure 4.13: Lifetime as a function of the event-rate with wakeup interval 1 sec for SMAC

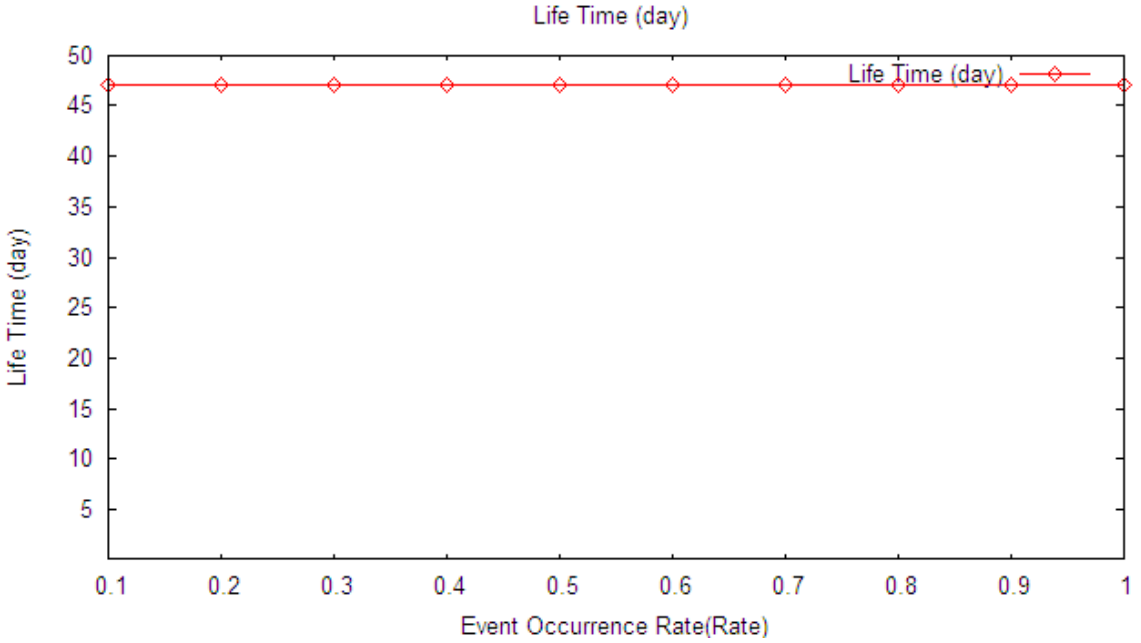

Figure 4.14: Lifetime as a function of the event-rate with wakeup interval 1 sec for TMAC

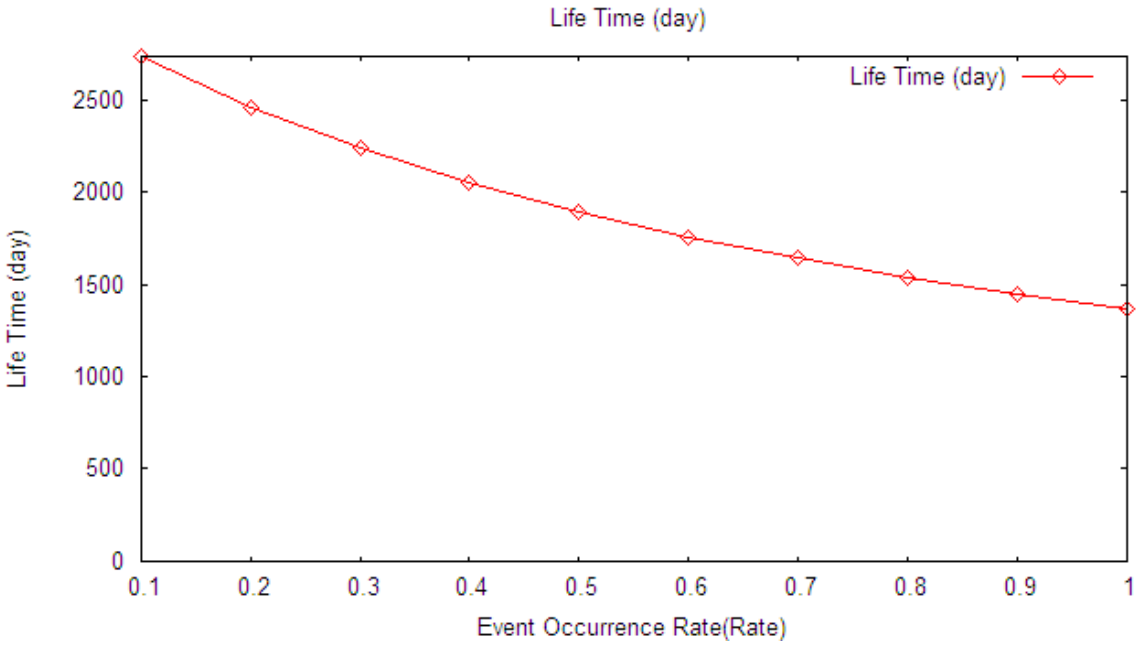

Figure 4.15: Lifetime as a function of the event-rate with wakeup interval 1 sec for BMAC

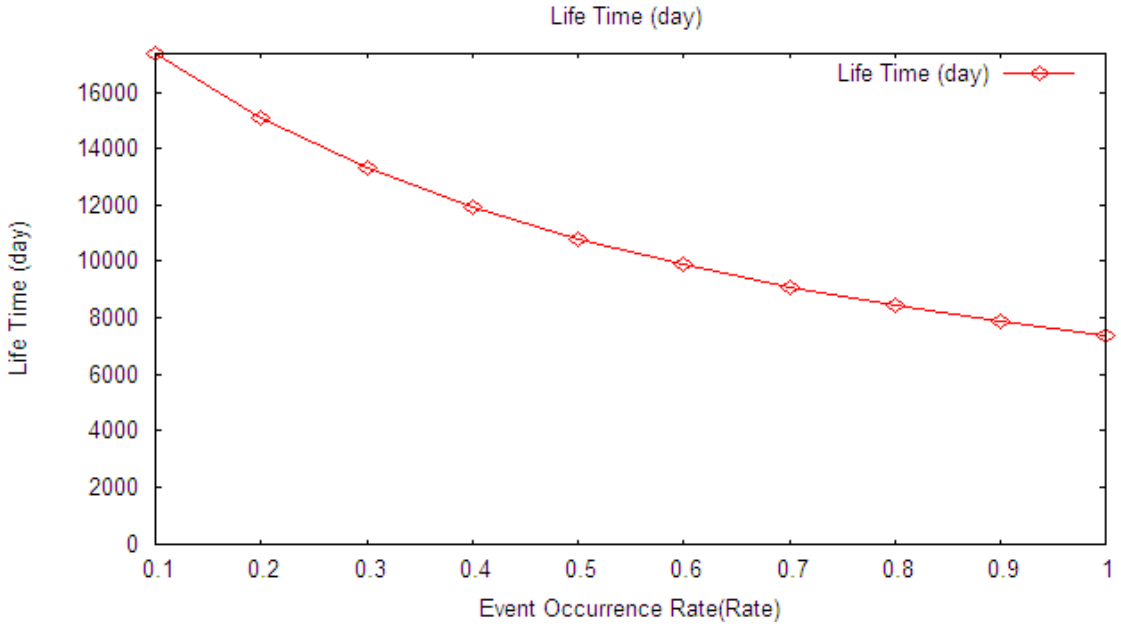

Figure 4.16: Lifetime as a function of the event-rate with wakeup interval 1 sec for XMAC

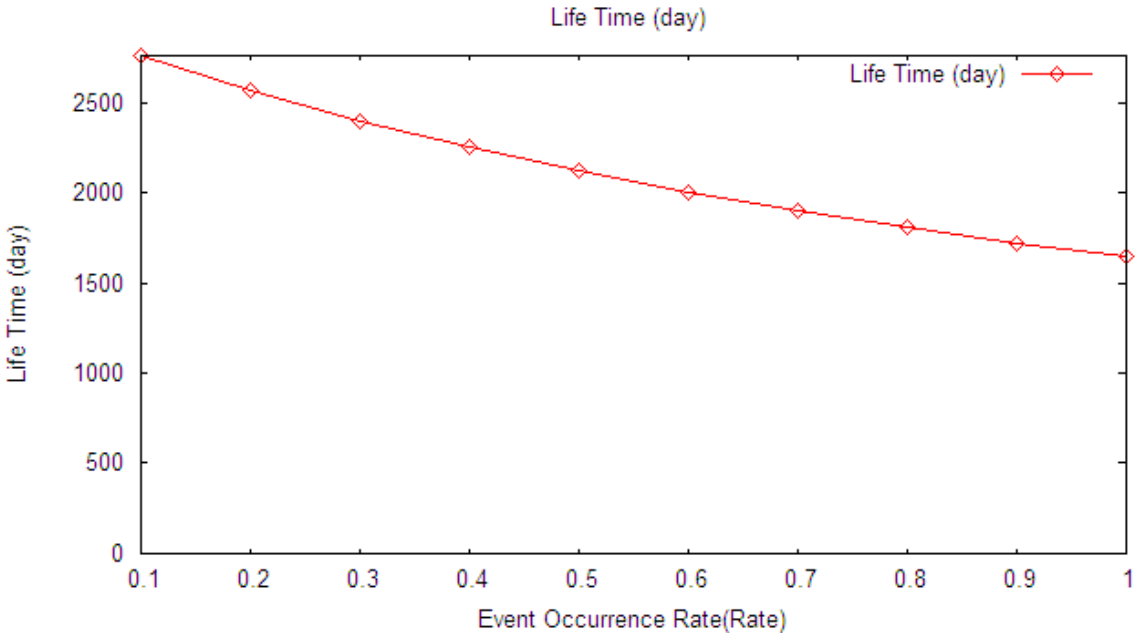

Figure 4.17: Lifetime as a function of the event-rate with wakeup interval 1 sec for SCPMAC

In the scenario, we assume that the application developer sets the per-hop delay requirement to 600 ms. Figure 4.18 provides the lifetime as a function of event-rate when the per-hop delay is 600 ms (assuming that the per-hop delay for every MAC protocol is equal to the wakeup interval). Because SCP-MAC results in the longest lifetime regardless of the event-rate, the developer can determine that SCP-MAC with 600 ms of wakeup interval is the best MAC protocol for this case.

In contrast, Fig. 4.19 provides the lifetime as a function of event-rate when the per-hop delay requirement is 15 seconds. When the event-rate is smaller than 0.5, SCP-MAC provides the best lifetime among all MAC protocols. However, when the event rate is larger than 0.5, T-MAC is better. Although the developer may not know accurately the event-rate, if the expected the event-rate to be smaller than 0.5, SCP-MAC should be chosen. If not, T-MAC. At this point developer has the optimal configuration of the networking stack meeting the application requirements. He can generate and compile the code for the application.

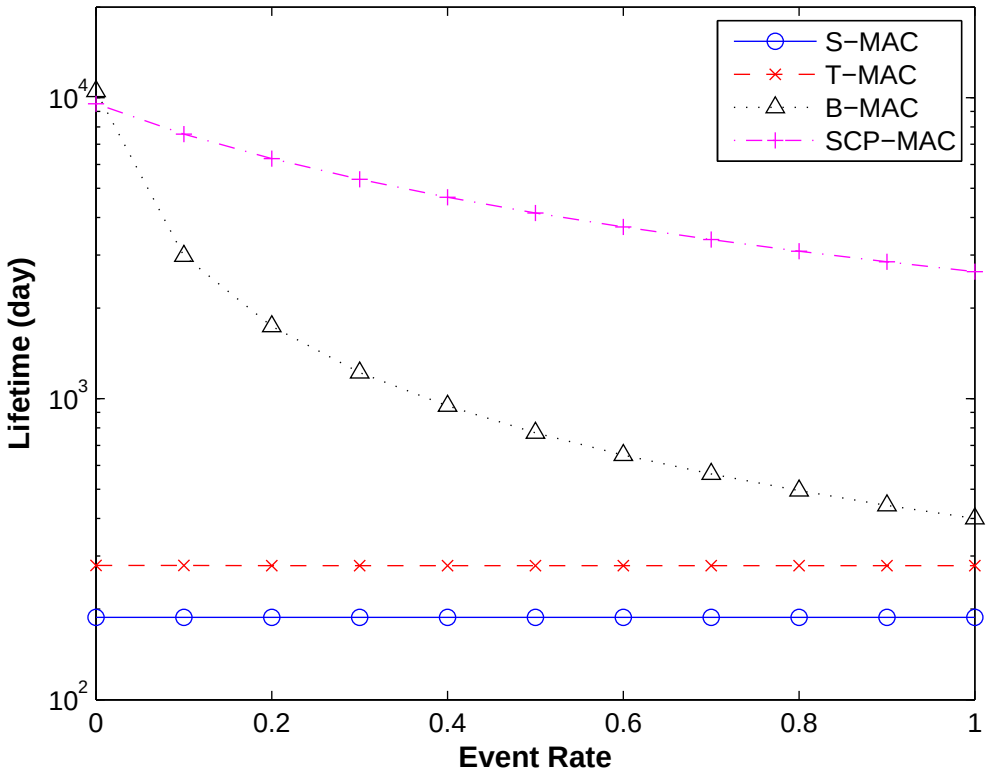

Figure 4.18: Lifetime as a function of event-rate with 600msec of the per-hop delay

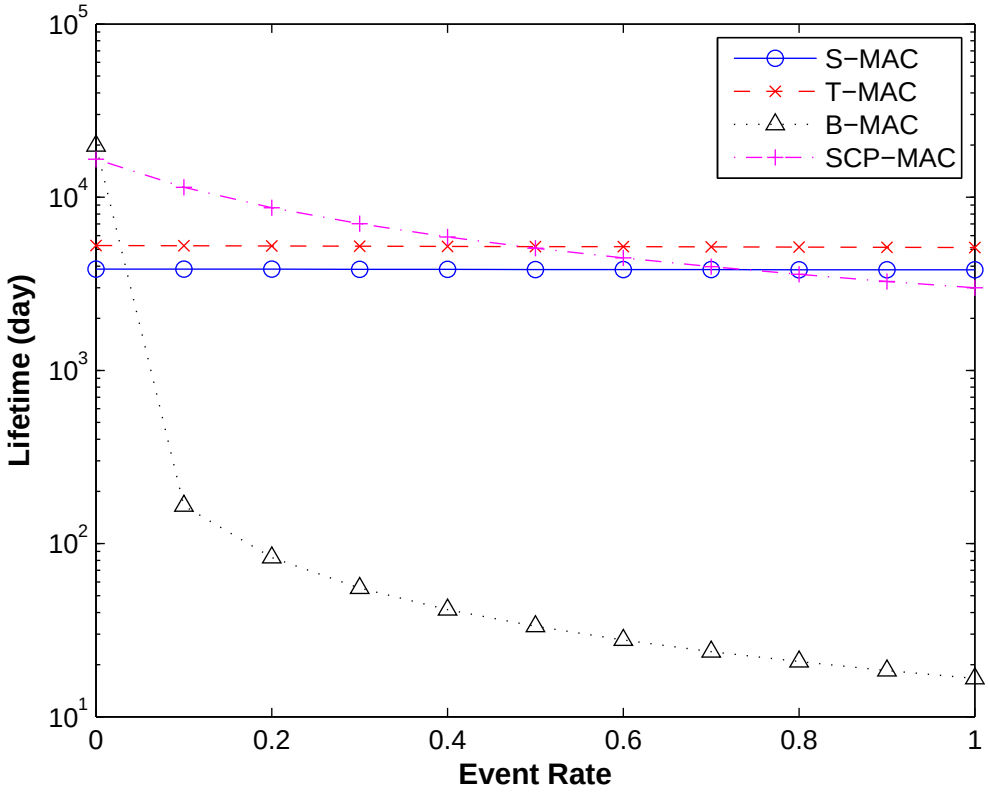

Figure 4.19: Lifetime as a function of event-rate with 15sec of the per-hop delay

Chapter 5

Support for Other Libraries

Although this version of RaPTEX supports nesC component for wireless sensor applications running over TinyOS, RaPTEX can work with other underlying system libraries. In this section, we discuss how to interface a new system library with the RaPTEX's GUI by focusing on the data representation scheme.

Since RaPTEX provides the block diagram based component composition environment, the functionality of the underlying library should be modularized and represented as a set of components by defining a library specific data representation scheme. A component in this context is a software element offering a pre-developed service and interfaces. A system is built by gluing prefabricated components together through predefined interfaces and configuring tunable variables. To include a set of components in RaPTEX, a library specific data representation scheme should contain some information about underlying system as follows:

- **Module's general description** : although this type of information depends on the library and code generator, in general, the code generator requires information such as component names and paths of related source files. Information such as whether the component contains main function or not also need to be included. In the case of the TinyOS library, the general descriptions contain a component name, a file path of a nesC source, version information and a flag whether the component is top-level or not.

- **Explicit input and output interfaces** : to connect components to each other and provide black box based component interactions, input and output ports in a component should be explicitly defined. An interface can be used for an invocation of a function call with the proper parameters between different modules. In the case of the TinyOS library, function calls between components are represented as nesC interfaces, which contain information such as commands (downward function call) and event handlers (upward function call).

- **Component level parameter** : a data representation scheme should contain information about tunable valuables to allow users to change behaviors of a component in the GUI. In the case of TinyOS library, component level parametersare defined using nesC annotation in nesC code directly, and the nesC XML contains these values.

- **Information about the interconnections between components** : because embedded communication systems in RaPTEX are basically built by composing existing components, gluing information between components should be explicitly saved in the data representation scheme. The code generator may use this information to generate the source code and build a complete system. In the case of TinyOS, a nesC XML file contains all involved interfaces with unique ID and wiring information with separate XML tag, and the nesC code generator takes this file as input to generate a nesC code.

Once a data representation scheme is decided for a given library, a proper adaptor class should be implemented to bridge between new data representation scheme and GUI, which can be done by extending base classes in GUI and overriding abstract methods such as SimpleDiagram, SimpleNode, SimpleLink, etc. By reading the data representation files, RaPTEX extracts outline information about the underlying system library and builds the component library tree and the block diagram. A library specific code generator takes the data representation file as user's configuration files to generate real system code. The code translation process should be convertible back and forth between user's configurations in GUI and real system codes based on a defined the data representation scheme.

Chapter 6

Conclusion

6.1 Conclusion

Currently it is far from trivial to design efficient embedded communication systems. To bridge the gap between the need and the difficulty, we provide RaPTEX, an IDE for embedded communication systems, which allows non-specialists to easily develop and customize the networking stacks and to quickly estimate their performance. In this thesis, we focus on WSN applications. RaPTEX provides three services. First, it provides a block diagram based GUI environment that facilitates component composition. Using this GUI, the application developer can assemble a sensor application by simply dragging, dropping, and wiring existing nesC components. Second, it features an automatic code generator with a parameter setting mechanism that can be used by the developers to further customize the protocol stack. Last but not least, it provides a performance estimation facility based on existing theoretical analysis, in which the developers can receive immediate feedback on impact of their design decisions. We present two case studies of developing WSN applications using RaPTEX.

6.2 Future Work

RaPTEX aims to facilitate the software development process for embedded communication systems. The current version has several limitation which we plan to address in the near future.

First, RaPTEX uses as the component representation scheme the nesC XML, which describes nesC components. Therefore, every nesC component in TinyOS source directory is (automatically) translated to the XML format to be used in the RaPTEX block diagram. However, because the translation is made by the nesC compiler, the component has to compile without errors and cannot be translated to XML code if not all dependencies are satisfied. A separate parse may avoid this drawback.

RaPTEX still require an application developer have some knowledge of TinyOS components and compositions. To support true non-specialists, we plan to provide higher programming abstractions such as template-based programming or highly configurable components.

Finally, advanced developers may expect much more advanced nesC editing facilities when developing nesC components. The current version of RaPTEX focuses on supporting wiring of existing components and estimate the performance of the composition. The development of new components must be done using a different environment (e.g., TinyDT or Eclipse TinyOS).

Bibliography

[1] I. Akyildiz, Y. S. W. Su, and E. Cayirci, "A survey on sensor networks," in *IEEE Communication Magazine, vol. 40, no.8*, August 2002, pp. 102–116.

[2] W. Ye, J. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 3, June 2004.

[3] T. van Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," in *Proc. of of the 1st ACM Conference on Embedded Network Sensor Systems (Sensys'03)*, 2003.

[4] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proc. of the 2nd ACM Conference on Embedded Network Sensor Systems (Sensys'04)*, 2004.

[5] W. Ye, F. Silva, and J. Heidemann, "Ultra-low duty cycle MAC with scheduled channel polling," in *Proc. of the 4th ACM Conference on Embedded Network Sensor Systems (Sensys'06)*, 2006.

[6] I. Rhee, A. C. Warrier, M. Aia, J. Min, and P. Patel, "Z-MAC: a hybrid MAC for wireless sensor networks," in *Proc. of the 3rd ACM Conference on Embedded Network Sensor Systems (Sensys'05)*, 2005.

[7] G.-S. Ahn, E. Miluzzo, A. T. Campbell, S. G. Hong, and F. Cuomo, "Funneling-mac: A localized, sink-oriented mac for boosting fidelity in sensor networks," in *Proc. of the 4th ACM Conference on Embedded Network Sensor Systems (Sensys'06)*, 2006.

[8] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-mac: A short preamble mac protocol for duty-cycled wireless sensor networks," in *Proc. of the 4th ACM Conference on Embedded Network Sensor Systems (Sensys'06)*, 2006.

[9] A. Woo, T. Tong, and D. Culler, "Taming the underlying chanllenges of relidable multihop routing in sensor networks," in *Proc. of The 1st ACM Conference on Embedded Networked Sensor Systems (Sensys'2003)*, November 2003, pp. 26–27.

[10] J. Aslam, Q. Li, and D. Rus, "Three power-aware routing algorithms for sensor networks," *Wireless Communications and Mobile Computing*, vol. 2, no. 3, pp. 187–208, Mar. 2003.

[11] Q. Li, J. Aslam, and D. Rus, "Online power-aware routing in wireless ad-hoc networks," in *MOBICOM*, July 2001, pp. 97–107.

[12] J.-H. Chang and L. Tassiulas, "Energy conserving routing in wireless ad-hoc networks," in *INFOCOM (1)*, 2000, pp. 22–31. [Online]. Available: citeseer.nj.nec.com/chang00energy.html

[13] "TinyOS community forum." [Online]. Available: www.tinyos.net

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 93–104.

[15] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The emergence of networking abstractions and techniques in TinyOS," in *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[16] D. Gay, P. Levis, R. V. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. of Programming Language Design and implementation(PLDI'2003)*, June 2003.

[17] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The design of an acquisitional query processor for sensor networks," in *In*

Proc. of (SIGMOD'03), San Diego, CA, June 2003. [Online]. Available: http://www.cs.berkeley.edu/~madden/acqp.pdf

[18] M. L. Sichitiu, "Cross-layer scheduling for power efficiency in wireless sensor networks," in *Proc. of Infocom 2004*, vol. 3, Hong Kong, PRC, Mar. 2004, pp. 1740–1750.

[19] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *In ACM International Workshop on Wireless Sensor Networks and Applications (WSNA02)*, September 2002.

[20] M. D. Yarvis, W. S. Conner, L. Krishnamurthy, A. Mainwaring, J. Chhabra, and B. Elliott, "Real-world experiences with an interactive ad hoc sensor network," in *In International Conference on Parallel Processing Workshops*, 2002.

[21] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, and J. H. vBruce Krogh, "Energy-efficient surveillance system using wireless sensor networks," in *Proc. the 2nd international conference on Mobile systems, applications, and services*, 2004.

[22] "ALERT." [Online]. Available: http://www.alertsystems.org

[23] "Shooter Localization." [Online]. Available: http://www.isis.vanderbilt.edu/projects/nest/applications.html

[24] "TinyDT: TinyOS plug–in for the Eclipse platform." [Online]. Available: http://www.tinydt.net/

[25] E. Cheong, E. A. Lee, and Y. Zhao, "Joint modeling and design of wireless networks and sensor node software," in *UCB/EECS-2006-150*, 2006, Technical Report.

[26] "GRATIS: Graphical development environment for TinyOS." [Online]. Available: http://www.isis.vanderbilt.edu/projects/nest/gratis/index.html

[27] S. Park, A. Savvides, and M. B. Srivastava, "SensorSim: A simulation framework for sensor networks," in *Proc. of Programming Language Design and implementation(PLDI'2003)*, June 2003.

[28] "The network simulator – ns–2." [Online]. Available: http://www.isi.edu/nsnam/ns/

[29] "OPNET." [Online]. Available: http://www.opnet.com/

[30] A. Varga., "The OMNet++ discrete event simulation system," in *Proc. of the European Simulation Multiconference (ESM'2001)*, June 2004.

[31] G. Simmon, P.Volgyesi, M. Maroti, and A. Ledeczi, "Simulation-based optimization of sommunication protocols for large-scale wireless sensor networks," in *Proc. of 2003 IEEE Aerospace Conference*, vol. 3, 2003, pp. 1339–1346.

[32] B. Greenstein, E. Kohler, and D. Estrin, "A Sensor Network Application Construction Kit (SNACK)," in *Proc. of The 2nd ACM Conference on Embedded Networked Sensor Systems (Sensys'2004)*, November 2004.

[33] P. Levis, M. W. N. Lee, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proc. of The 1st ACM Conference on Embedded Networked Sensor Systems (Sensys'2003)*, 2003, pp. 126–137.

[34] P. Baldwin, S. Kohli, and E. A. Lee, "Modeling of sensor nets in Ptolemy II," in *Proc. of Information Processing in Sensor Networks (IPSN'2004)*, April 2004, pp. 26–27.

[35] "MVC." [Online]. Available: http://heim.ifi.uio.no/~trygver

[36] D. Gay, P. Levis, D. Culler, and E. Brewer, "nesC 1.2 language reference manual."

[37] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Z. (eds.), "Heterogeneous concurrent modeling and design in Java: Volume 3: Ptolemy ii domains," in *Technical Report Technical Memorandum UCB/ERL M05/23*, July 2005.

[38] E. A. Lee and S. Neuendorffer, "MoML - a modeling markup language in XML - version 0.4," University of California at Berkeley, Tech. Rep., March 2000. [Online]. Available: http://www.gigascale.org/pubs/16.html

[39] "Code generation network." [Online]. Available: http://www.codegeneration.net/

[40] D. G. P. Levis and D. Culler, "Software design patterns for TinyOS," in *LCTES'05: ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, June 2005.

[41] S. Yoon and M. L. Sichitiu, "Analyzing power consumption of wireless sensor networks mac protocols," NCSU/Wireless Ad-hoc and Local Area Networks Research Lab," Technical Report, 2006.