Model-based Development for MAC Protocols in Industrial Wireless Sensor Networks

Admar Ajith Kumar Somappa[1,2] and Kent Inge Fagerland Simonsen[1]

[1] Bergen University College, Bergen, Norway
[2] University of Agder, Grimstad, Norway
{aaks,kifs}hib.no

Abstract. Model-Driven Software Engineering (MDSE) is an approach for design and implementation of software applications, that can be applied across multiple domains. The advantages include rapid prototyping and implementation, along with reduction in errors induced by humans in the process, via automation. Wireless Sensor Actuator Networks (WSANs) rely on resource-constrained hardware and have platform-specific implementations. Medium Access Control (MAC) protocols in particular are mainly responsible for radio communication, the biggest consumer of energy, and are also responsible for Quality of Service (QoS). The design and development of protocols for WSAN could benefit from the use of MDSE. In this article, we use Coloured Petri Nets (CPN) for platform independent modeling of protocols, initial verification, and simulation. The PetriCode tool is used to generate platform-specific implementations for multiple platforms, including MiXiM for simulation and TinyOS for deployment. Further the generated code is analyzed via network simulations and real-world deployment test. Through the process of MDSE-based code generation and analysis, the protocol design is validated, verified and analyzed. We use the GinMAC protocol as a running example to illustrate the design and development life cycle.

Keywords: Model-Based Development, Code-generation, Medium Access Control Protocols, Colored Petri Nets, Simulation, Implementation, Wireless Sensor Actuator Networks (WSAN)

1 Introduction

A wireless network of connected sensors and actuators operating to fulfill a specific collective goal constitutes a Wireless Sensor-Actuator Network (WSAN). The sensors and actuators are resource-constrained devices operating on batteries. Among several application domains, process automation and factory automation are important application areas in the industrial domain. Specifically, the application of a WSAN in control-loop automation is an important research area. These applications have strict real-time requirements and are in many cases safety critical (e.g., nuclear power plants). Thus, the design of solutions that include software for the network nodes is required to have sound design and development methodology to result in a verified and validated design that also satisfies the real time requirements. The classical design methodology is to: (1) outline the requirements to the solution; (2) design a solution based on the requirements; (3) carry an analytical evaluation of the performance of the solution; and (4) do a manual conversion of the design into simulation code for further performance analysis, and (5) convert the design into implementation code and perform deployment test on hardware.

The Dual-Mode Adaptive MAC protocol (DMAMAC) [10] is a Medium Access Control (MAC) protocol for process control applications. The traditional design methodology was employed for DMAMAC protocol design [10] based on application requirements and further evaluated analytically. The DMAMAC protocol was simulated and evaluated for performance [9]. Further, was evaluated with real-world deployment [11] (these steps correspond to 1,2,3,4 & 5). Three important issues that can arise in the general design methodology are: human induced errors in manual conversion, time consuming manual conversion, and the requirement to make manual changes at each step when changes are required to the design or the requirements. With the use of emerging software engineering practices, one can improve the design and development process. This could help in further strengthening the reliability of the software part of the solution, while additionally reducing the time from design to development, thus reducing the cost. Model-Driven Software Engineering (MDSE) [3] is one such approach that has long been seen as a prominent approach for software engineering. MDSE is currently used in several industrial application domains [6]. In this article, we attempt to create a MDSE approach with MAC protocols in focus. The DMAMAC protocol is rather complex compared to the GinMAC protocol upon which the DMAMAC protocol design is based. Thus, we use the GinMAC protocol as a basis to build the MDSE approach and then proceed towards applying the principle to the DMAMAC protocol as well.

In the MDSE approach, we start with an abstract platform independent representation of the solution, protocols for example. Abstraction allows for focusing on behaviour of the protocol. Using formal approaches on the abstract models allows for verification of the behaviour of the protocol via model checking or theorem proving. This abstract model can further be simulated to obtain an initial performance assessment. Thus, the protocol can be validated for performance requirements, and verified for software requirements. One tool that allows both model-checking and simulation is CPN Tools [7]. CPN Tools is based on the expressive Colored Petri Nets (CPN) language combined with the Standard ML programming language. Previously, CPN has been used for modeling and verification of network protocols [1]. Further, we use Petricode [20] tool for the semi-automatic code generation part. This forms the MDSE approach proposed previously in [12]. In this article, we extend this work to complete the code generation for the simulation platform and hardware platform. The generated code is used to analyze the protocol performance via network simulations on MiXiM, and via deployment in a real-world setting of the TinyOS code. We also discuss the methodology used to design the CPN model. We use GinMAC [21] protocol as a running example to present our MDSE approach.

Related Work. A model-based development approach has been applied in the WSAN domain [15, 22, 19]. Multiple works have proposed frameworks for rapid prototyping of the development model, mostly based on either, Domain Specific Modeling Languages (DSML) [2, 4] or the Unified Modelling Language (UML) [15, 22, 19]. In [16], the authors propose a design framework to convert models created in Simulink to platform specific code for the platforms TinyOS and MANTIS operating system. They also provide simulation and behavioural analysis. An Architectural framework for Wireless Sensor Actuator Networks (ArchWiSeN) was proposed in [18]. This is based on the generic modeling platform, UML, for abstract and platform independent representation of the models. Platform specific code generation is performed to obtain code for TinyOS, and simulated using the Micaz platform provided in the TOSSIM [14] simulator by TinyOS.

In this article, we choose to create the platform-independent abstraction of the model in CPN Tools. CPN tools allows state-space based verification, and simulation. We specifically focus on behavioural modeling of the protocol design. A typical WSN node implements an application protocol, a routing protocol, a MAC protocol, and a link layer protocol. Thus, the solution in its entirety is made up of multiple protocols, making up a complex solution. In [16, 18], the authors view the solution as multiple nodes depicting common WSN behaviour. Further, in this article, we provide platform specific code generation for MiXiM, a network simulator specifically built for wireless networks. Also, the deployment specific code generation targets the TinyOS platform and is tested via deployment on Zolertia Z1 [23] motes.

The rest of the paper is divided into five sections. We revisit our model-based development approach and PetriCode [20] in Section 2. Also, the GinMAC protocol [21] which is used as an example is introduced. In Section 3, we describe the CPN model of the GinMAC protocol. The code generation templates, pragmatics used, and the generated code for MiXiM-OMNeT are discussed in Section 4. Code generation for TinyOS is discussed in Section 5. Finally, in Section 6, we sum up conclusions and discuss future work. The article assumes prior knowledge of Petri nets. The MDSE approach and Model-based Development approach are used interchangeably, and means the same in this context.

2 MDSE and PetriCode

The MDSE approach is shown in Fig. 1. Compared to the MDSE approach presented in the previous article [12], we have realized the work in progress (now the working part), and have implemented and analyzed the generated code via simulation and deployment. This includes the design of an example

MAC protocol, code generation for the simulation platform MiXiM [8] based on OMNeT, and for the operating system TinyOS [13] for hardware platforms. The GinMAC [21] protocol is used as a running example for the code generation case study and is introduced later in this section. Following the MDSE approach the GinMAC protocol is first designed using CPN Tools. Initial design verification, validation, state-space analysis, and simulation are performed using CPN Tools. Further, platform dependent code generation is performed to obtain code for MiXiM and TinyOS. The code generation is done using the PetriCode tool as described below.

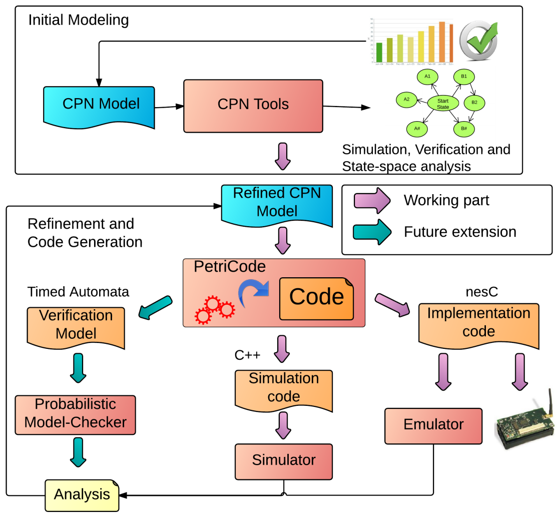

Fig. 1. MDSE approach for protocol design [12]

PetriCode. We use the PetriCode [20] tool for the code generation. PetriCode is a template based code generation tool designed to transform a subclass of CPN models called Pragmatic Annotated-CPNs (PA-CPN) to implementations. PA-CPN models enforce a hierarchical structure on the models with three levels. The top level module in a PA-CPN model is called the Protocol System Module (PSM). The PSM contains the principal agents of the protocol and the channels between them. Each principal in the PSM contains sub-modules that are on the principal level. The principal level modules (PLMs) contain the services that are provided by the principal as well as places that are common among the services of the principal, and life-cycle variables that control when services can be invoked. Each service in the PLMs has sub-modules on the service level. Service level modules (SLMs) contain the actual behaviour of each service. SLMs are further decomposable into control-flow blocks that represent structures such as loops and conditionals. In addition to enforcing a hierarchical structure, PA-

CPNs allow model elements to be annotated with code generation pragmatics. The pragmatics allow the code generator to identify the elements of the PA-CPN model and to find appropriate templates for the code generation. The code generation process in PetriCode starts by reading and parsing a PA-CPN model. Then the PA-CPN model is transformed into an intermediary representation called an Abstract Template Tree (ATT). The ATT mirrors the hierarchical structure of a PA-CPN model. PetriCode exploits the pragmatics, to decide what templates to execute at each node in the ATT. The generator executes templates for each of the nodes in the ATT. Finally, the code generated for the ATT nodes are combined to obtain the complete code for each principal. In this article, we design and develop separate templates for the MiXiM and TinyOS platforms. Based on these templates the final platform dependent code is obtained.

Code Generation Goals. The code generation phase assists in reducing errors induced by humans in programming and streamlines the implementation approach based on a modular implementation approach. Also, the conversion to simulation platform assists in analyzing the design of the protocol in the simulated world to assess the performance. Network simulators specialized in wireless simulations offer emulated environments of wireless channels, energy consumption, data transmission, and other services. The conversion to MiXiM code for simulation has two advantages: firstly, performance analysis of the designed protocol; secondly, comparison of the protocol with existing protocols based on selected performance metrics. The conversion to TinyOS supporting Network Embedded Software C (nesC) code allows the protocol to run across multiple hardware components that support the TinyOS platform. This also allows the users to validate the operation of their protocol on existing hardware, and provides opportunities for further testing in a real environment.

2.1 The GinMAC Protocol

GinMAC is a Time Division Multiple Access (TDMA) protocol, proposed in the GINSENG project [17]. GinMAC was developed to address the real-time requirements for industrial monitoring and control applications. A typical GinMAC superframe is shown in Fig. 2(b). A Finite State-Machine (FSM) representation of a node working on the GinMAC protocol is shown in Fig. 2(a). The main features of the protocol are: *Offline Dimensioning*, *Exclusive TDMA*, and *Delay Conform Reliability Control. Offline Dimensioning* means that deployment and delay requirements are planned before deployment, and all scheduling decisions are made offline prior to deployment. *Exclusive TDMA* means that all TDMA slots are exclusive and are not re-used. *Delay Conform Reliability Control* means that GinMAC uses reliability control mechanisms in the form of additional re-transmission slots to increase reliability. The number of additional slots required are calculated based on the wireless channel conditions in the area of deployment. The GinMAC protocol is designed for networks having a tree topology. It has three types of slots: *basic*, *additional*, and *unused*. *Basic* slots are for regular sensor and actuator data. The *Additional* slots feature is intended to increase robustness in poor channel conditions. *Unused* slots are sleep slots for duty cycling and energy conservation.

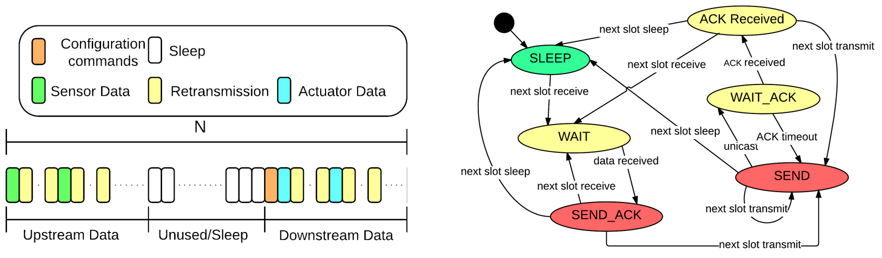

Fig. 2. The GinMAC superframe (a) and the GinMAC Finite State Machine (b)

3 CPN Model

As a first step we create a CPN model for the GinMAC protocol. The top level module (Protocol System Module) of a sensor/actuator node within the GinMAC protocol is shown in Fig. 3. The platform independent CPN model includes an application/network module, the GinMAC module, a radio module, and a wireless channel module for data exchange. The pragmatics assisting in the code generation are by convention written inside <<>>.

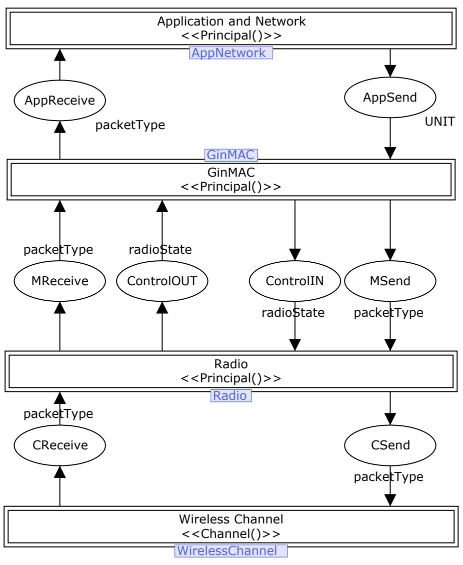

Fig. 3. Top-level CPN module

3.1 The GinMAC Module

We use a modular modeling approach for design. Different MAC protocols share common features that can be designed as basic re-usable components which further increases the re-usability of protocol design and also the code generation approach. The abstract model in CPN is platform independent, hence the model can be used to generate code-specific to different platforms as well. The re-usability

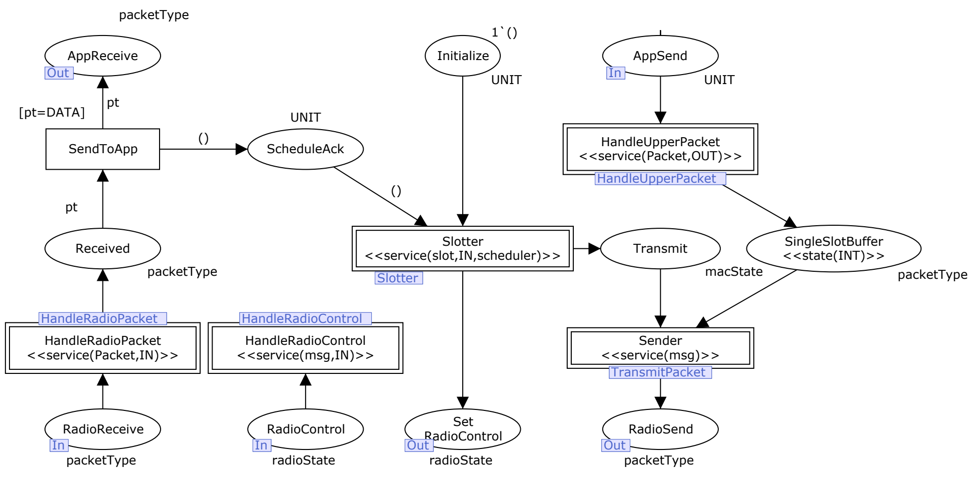

Fig. 4. CPN model of a MAC protocol

allows different MAC protocols to be modeled and code to be generated for the same platform. The detailed GinMAC layer in the principal (second) level is shown in Fig. 4. This model contains the GinMAC functions including: send/receive data packets, slot scheduler, and control packets handler. The send/receive of data packets facilitates the data transfer for the application through the entire network and is handled by two services: Sender and HandleRadioPacket. The radio control packet handling is local to the node and is handled by the service HandleRadioControl. The slot scheduler manages the operation to be performed at the node at each instant of time based on the superframe. The slot scheduler service is handled by Slotter. The main set of operations in GinMAC also called as MAC states includes: send, wait (receive), and sleep as depicted in the FSM in Fig 2(a). These operations of the GinMAC protocol are grouped as services which are defined in more detail in the service level of the PetriCode approach. The operations defined here are also basic MAC operations for other protocols including the DMAMAC protocol, and hence can be re-used.

3.2 The Slotter Function

Fig. 5 shows the *slotter* function of the protocol used to implement different slots based on the timer. These slots represent one of the operations to be performed. Since GinMAC is a TDMA-based protocol, a timer is used to run through the slots. The *slotter* function calls services based on the operations: send (SEND_DATA/ACK), receive (WAIT_DATA/ACK), and sleep, appropriately depending on the time. Based on the slot type, an appropriate service is called to handle the operation: Receive, Transmit, or Sleep. In Fig. 5, we have omitted send and wait for notifications to keep the figure simple and small, but the design of notification slot is included in the complete model and in the code generation templates. Notifications are an alternative representation of the configuration commands used by GinMAC protocol, a generic concept used by MAC protocols to send network wide updates.

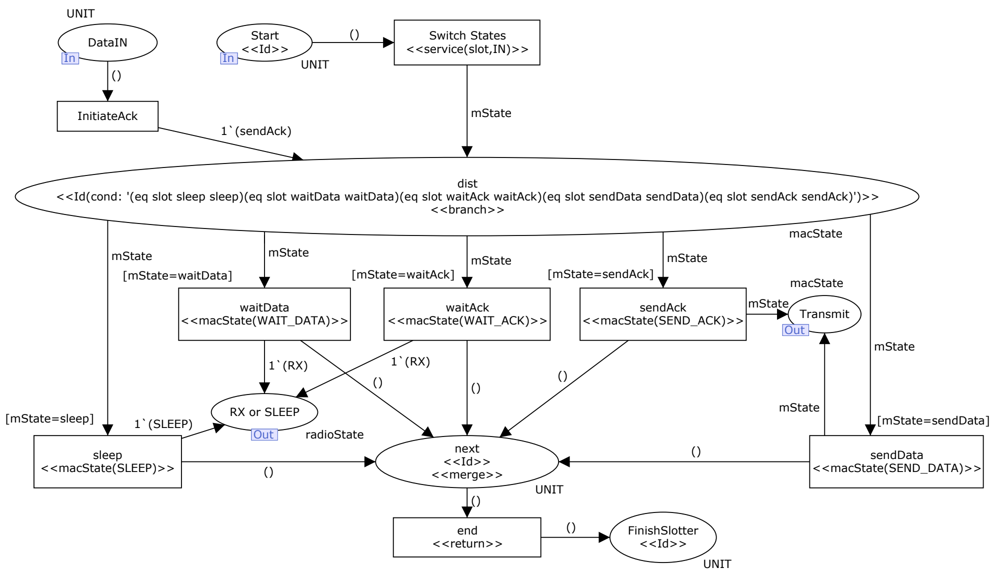

Fig. 5. Slotter function of the MAC protocol

One of the services used by Slotter, the Sender service for transmitting packets is shown in Figure. 6. The service handles all outgoing packet types including data, ack, and notification packets. It sends the packet to the radio service for further handling of the physical transmission.

4 MiXiM Code Generation

OMNeT++ is graphical modeling framework along with discrete event-based simulation and analysis extensions with graphical user interface support. MiXiM [8] is a modeling framework, which is a result of integrating several OMNeT++ frameworks, designed specifically for simulating mobile and fixed wireless networks. MiXiM is a modular framework consisting of pre-installed implementation code for radio modules, and base modules for application, network (routing), and the MAC layer. The workflow for simulation-analysis including the node software architecture in MiXiM is shown in Fig. 7.

MiXiM simulations are specified using a combination of source files, NED files (Network Descriptors), configuration.ini (configuration file), and other XML files describing the physical attributes of the network. The source C++ files are mainly used describe the protocols at each layer. The NED files are used to

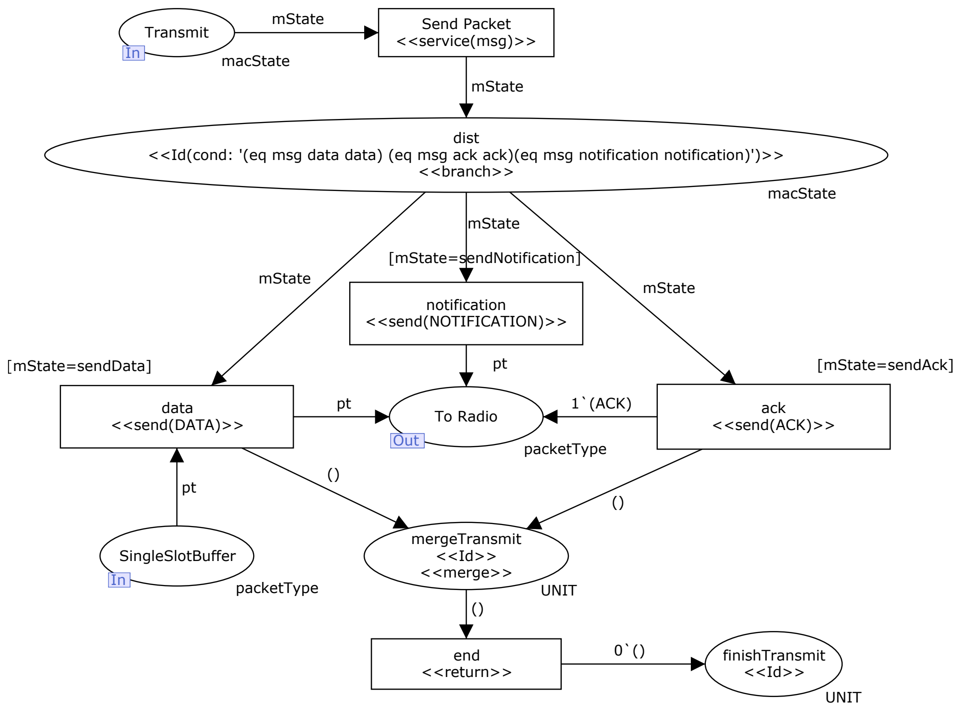

Fig. 6. Transmit Packet function of the GinMAC protocol

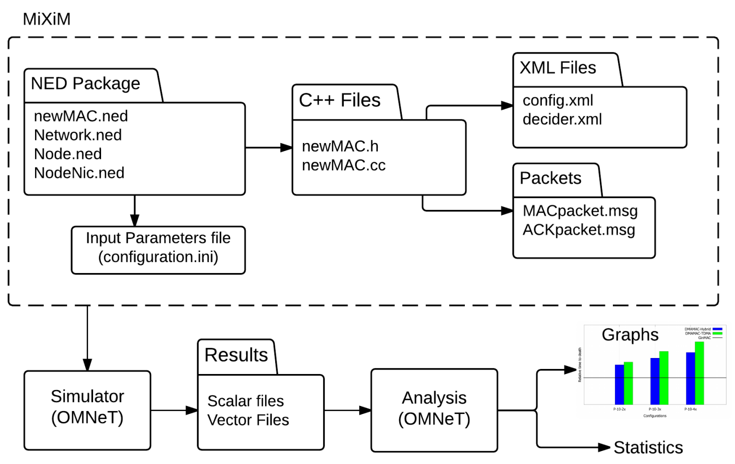

Fig. 7. MiXiM node software architecture with simulation and analysis flow

describe each type of physical node in the network, the network configuration defining how these nodes connect with each other, and the user defined module to be used in these nodes. The configuration.ini file allows the user to define explicitly various parameters concerning each layer (physical, MAC, application and network). It also includes position data for nodes (if required) and multiple configurations for simulation. Further, XML files are used to define the path-loss models, packet loss, signal to noise ratio (SNR), and decider configurations. These features are specific to the MiXiM simulation platform and are generated for basic configuration using the code generation templates, and are not a part of the CPN model. Apart from the generation of the MAC source files, NED files for basic network layer configuration, basic configuration files, and network configuration XML files are also generated.

4.1 MiXiM MAC model

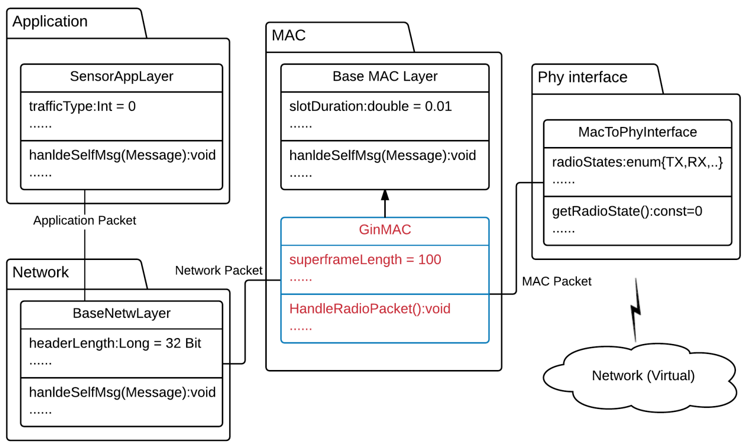

Fig. 8. An abstract UML view of the source code for a node

The focus is to generate the MAC layer implementation code, and hence we use the existing modules of the MiXiM framework to construct the complete implementation. Creating a new module in MiXiM includes implementing the base functions for that module (BaseMacLayer) and extending it with the protocol specific functions. We use the code templates to generate the protocol with methods extended from the BaseMacLayer. The generated MiXiM source code from the protocol model is then used as the MAC protocol, and other modules (application/routing) are used to complete the node software architecture to make it simulation ready. An abstract UML representation of the generated source code (GinMAC) put in context along with the existing modules is shown in Fig 8. The GinMAC source code generated for MiXiM inherits from the BaseMacLayer, and extends it with its functions and characteristics. The functional view of the GinMAC protocol is shown in Fig. 9. The functional view corresponds to the principal level of the CPN model shown in Fig. 4. The MAC protocol handles application/network packets and incoming packets from other nodes, and sends them over the radio channel to a corresponding node based on a pre-decided schedule determined by the GinMAC superframe structure shown in Fig. 2(b).

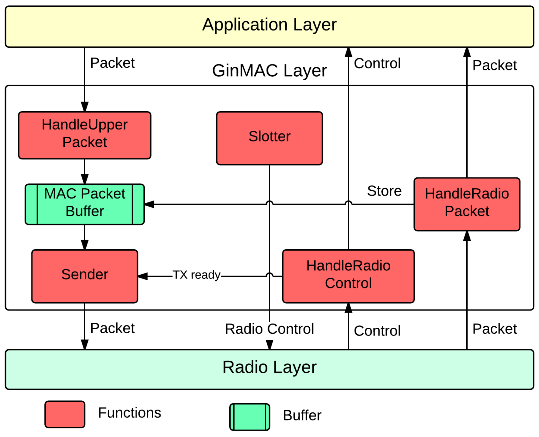

Fig. 9. A graphical representation of the functions in the MiXiM MAC protocol

4.2 PetriCode

The MAC protocol designed in CPN is used to generate a MiXiM equivalent. We use the PetriCode tool [20] for the code generation. The PetriCode tool, based on the code generation templates developed for the MiXiM platform, generates the required C++ source code. An example template for the GinMAC states SEND_DATA, WAIT_DATA, and SLEEP are presented in List. 1.1. The main GinMAC states are: SEND_DATA, SEND_ACK, SEND_NOTIFICATION, WAIT_DATA, WAIT_ACK, WAIT_NOTIFICATION and SLEEP. A MAC state represents the type of operation being performed at a given time. Each of the MAC states have an associated code template for generation. Based on these code templates, we obtain the main source code. The code template example for MAC states is shown in List. 1.1. The params variable in the template checks for the parameters within the keyword (macState) for the associated template, then generates the code for it. The generated C++ source code for the Slotter function shown in Fig. 5 is presented in Listing 1.2. Three of the MAC states SEND_DATA, WAIT_DATA and SLEEP have been presented along with the generated code. In the full version implemented, all the MAC states have their respective code templates as opposed to only three shown here.

Listing 1.1. PetriCode C++ code generation template

```
Pragmatic : <macState(SLEEP)>
Template code for MiXiM:
<%if(params[0].toString() == "SLEEP")
{%> currentMacState = SLEEP;
    debugEV << "Going to Sleep" <<endl;
    phy->setRadioState(MiximRadio::SLEEP);
    /* @brief Finds the next slot after getting up */
    findDsitantNextSlot();
<%}

Pragmatic : <macState(WAIT_DATA)>
Template code for MiXiM:
<%if(params[0].toString() == "WAIT_DATA")
{%> currentMacState = WAIT_DATA;
        debugEV << "My receive slot" <<endl;
        phy->setRadioState(MiximRadio::RX);
        debugEV << "Switching Radio to receive mode" << endl;
        /* @brief Procedure for finding nextSlot */
        findImmediateNextSlot(currentSlot, slotDuration);
        currentSlot++;
        currentSlot %= numSlots;
<%}

Pragmatic : <macState(SEND_DATA)>
Template code for MiXiM:
<%if(params[0].toString() == "SEND_DATA")
{%> currentMacState = SEND_DATA;
    if (mySlot == transmitSlot[currentSlot]){
        if(macPktQueue.empty()){
            debugEV << "No Packet to Send exiting" << endl;
            findNextSlot(currentSlot, slotDuration);
    }else{
        phy->setRadioState(MiximRadio::TX);
        debugEV << "Waking up in my slot.
            Radio switch to TX command Sent" << endl;
        findImmediateNextSlot(currentSlot, slotDuration);
    }
    currentSlot++;
    currentSlot %= numSlots;
<%}
```

Listing 1.2. C++ source code

```
//GinMAC C++ file
void GinMAC::Slotter(message_t msg){
if( msg == sleep){
    currentMacState = SLEEP;
    debugEV << "Going to Sleep" <<endl;
    phy->setRadioState(MiximRadio::SLEEP);
    /* @brief Finds the next slot after getting up */
    findDsitantNextSlot();
}
else if( msg == waitData){
    currentMacState = WAIT_DATA;
    debugEV << "My receive slot" <<endl;
    phy->setRadioState(MiximRadio::RX);
    debugEV << "Switching Radio to receive mode" << endl;
    /* @brief Procedure for finding nextSlot */
    findNextSlot(currentSlot, slotDuration);
    currentSlot++;
    currentSlot %= numSlots;
}
else if( msg == sendData){
    currentMacState = SEND_DATA;
    if (mySlot == transmitSlot[currentSlot]){
        if(macPktQueue.empty()){
            debugEV << "No Packet to Send exiting" << endl;
            findNextSlot(currentSlot, slotDuration);
    }else{
        phy->setRadioState(MiximRadio::TX);
        debugEV << "Waking up in my slot.
            Radio switch to TX command Sent" << endl;
        findImmediateNextSlot(currentSlot, slotDuration);
    }
    currentSlot++;
    currentSlot %= numSlots;
}}
```

4.3 MiXiM Simulation

The MiXiM simulation engine allows for simulation, and statistics collection, which can be further used to generate graphs. The graphs support the assessment of the performance of the protocol. Statistics can be collected in either scalar or vector form. For our simulation test we used the scalar forms. We performed test simulation on a small network with 7 nodes and 1 sink. The node configuration used for the study is shown in Fig. 10. We used the CC2420 radio decider module, the CC2420 radio power consumption values, and switch times to obtain accurate results.

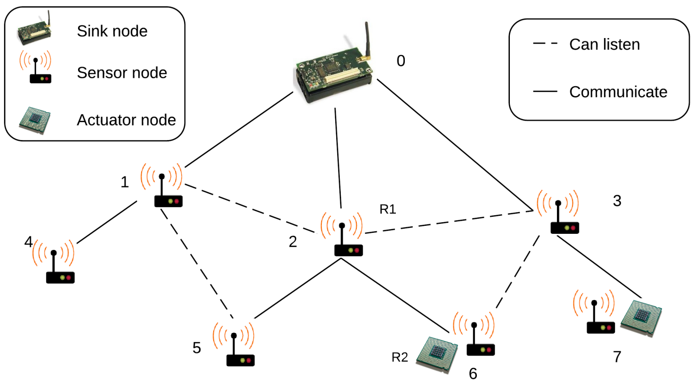

Fig. 10. The network topology used for simulation

The resulting graph from counting the number of packets transmitted and received for each node is given in Fig. 11. Also, a graph with network lifetime based on a given initial battery capacity is shown in Fig. 11. The lifetime is related to the starting capacity. Given the load, it is evident that node 2 depletes

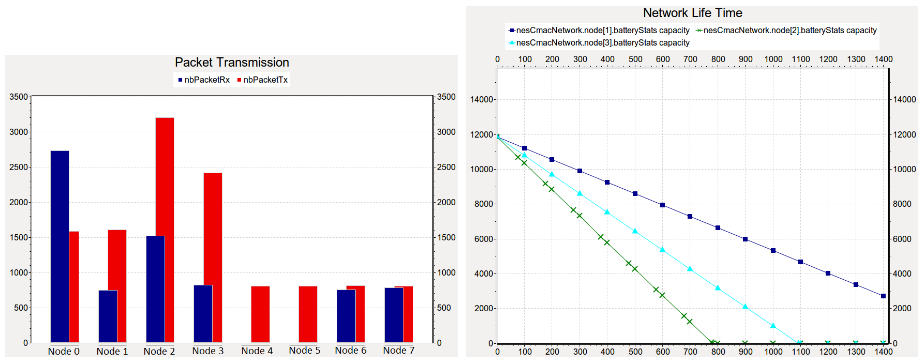

Fig. 11. The packet reception and transmission statistics for the GinMAC protocol (a) and Network life time in terms node deaths (b)

its energy first among all the nodes in the network and thus is the first to die as shown in Fig. 11. The data transmission graph shows the amount of data transmission being handled for a given simulation duration, 1500 seconds for our case. In the process two nodes die, node 2 and node 3.

5 TinyOS nesC Code Generation

TinyOS is one of the commonly used operating systems for resource limited WSAN hardware implementations. nesC is a component-based programming language for the TinyOS platform. It mainly consists of *components* which are represented using *modules* and *configurations*. These components generally provide services to other components and use functions provided by other components via a set of *interfaces*. The implementation part uses *commands* and *events* which are called or signaled, respectively. *Commands* are used to

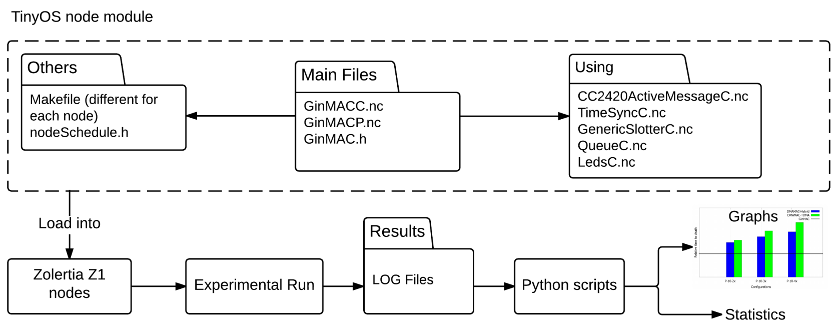

Fig. 12. nesC implementation, deployment and analysis flow

define operations that can be triggered. *Events* represent hardware events that is similar to an interrupt to indicate an event occurring, e.g. reception of a packet. *Tasks* are also a part of the programming language which can be called from both events and commands. The workflow for implementation on TinyOS using the generated nesC source code, and further performing deployment and analysis is shown in Fig. 12. For evaluation of the generated nesC code the presented workflow is used. The generated source code along with an application is compiled for the platform hardware Zolertia Z1 [23]. Furthermore, based on pre-decided topology, schedule, and location, we conducted an experiment. The collected results are placed in a log file, which is used by python scripts to generate graphs and statistics. The generated GinMAC source code uses multiple off the shelf components to complete the solution, e.g., the CC2420ActiveMessageC radio module provided by TinyOS.

5.1 MAC nesC Model

The component graph of the GinMAC nesC module is shown in Fig. 13. The GinMAC protocol uses the radio functions provided by the existing component in TinyOS, the CC2420ActiveMessageC component. For time synchronization, an essential function required for Time Division Multiple Access (TDMA) protocols like GinMAC, we use the TimeSyncC component. The TimeSyncC provides the Flooding Time Synchronization Protocol (FTSP). As a scheduler, the GinMAC component uses the GenericSlotterC component. The scheduler goes through the superframe structure and executes the corresponding slot for the given instance. The component QueueC is used to create a packet buffer for incoming packets to be forwarded. The incoming arrows to the GinMAC module are the features that are provided by the GinMAC module to the application or the network modules that use GinMAC. In the current version, we generate code that uses the radio layer in an abstract form. Whereas a more detailed and explicit control of radio can be implemented similar to the MiXiM code. This is detailed in the implementation of MAC layers in TinyOS 2 [5]. With such explicit control, a MAC protocol can control the transmission power at which each packet is sent.

5.2 PetriCode

Similar to the MiXiM code generation, the GinMAC protocol designed in CPN is used for nesC code generation targeting TinyOS. PetriCode, based on the templates defined for nesC code generation, generates the nesC code. We present the same three GinMAC states and the template for code generation as in MiXiM. The GinMAC states SLEEP, WAIT_DATA and SEND_DATA are presented along with the employed code generation templates as shown in List. 1.3. The generated nesC source code for the Slotter function shown in Fig. 5 is presented in List. 1.4.

Listing 1.3. PetriCode nesC Template

```
Pragmatic: macState<SLEEP>
```

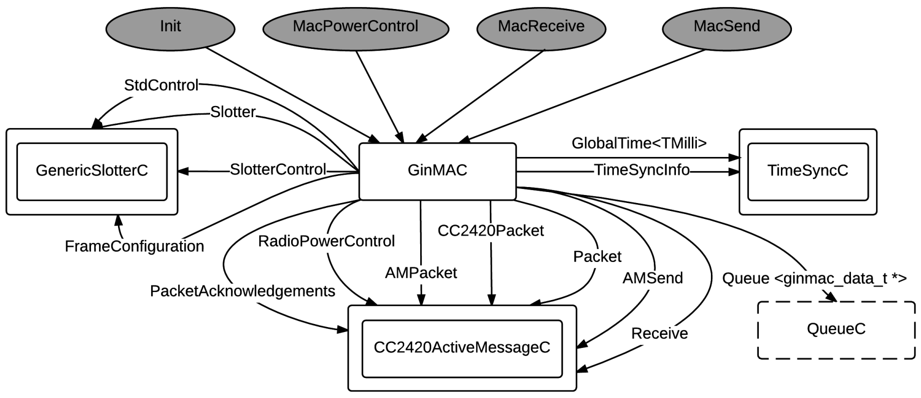

Fig. 13. GinMAC protocol nesC component model

```
Template code for nesC:
if(params[0].toString() == "SLEEP")
{%>     currentMacState == "SLEEP";
        if(!radioOff){
                printfz1("Calling radio sleep");
                call RadioPowerControl.stop();
                call Leds.led0Off();
}<%}%>

Pragmatic: macState<WAIT_DATA>
Template code for nesC:
if(params[0].toString() == "WAIT_DATA")
{%>     currentMacState == "WAIT_DATA";
        printfz1("Waiting for Data");
        /* @brief Switching on Radio if OFF, there should be an event if packet is arrived */
        if(radioOff){
                call RadioPowerControl.start();
                call Leds.led0On();
}<%}

Pragmatic: macState<SEND_DATA>
Template code for nesC:
else if(params[0].toString() == "SEND_DATA")
{%>     currentMacState == "SEND_DATA";
        /* @brief Checking if the slot is the node's transmit slot  */
        data = (data_t*)call Packet.getPayload(&dataPkt,sizeof(data_t));
        data->nodeId = TOS_NODE_ID;
        data->destinationId = sinkId;
        data->dataSeqNo = dataSeqCount;
        if (phyLock == FALSE){
                if(radioOff){
                        /* @brief Switching on the Radio */
                        printfz1("Waking up");
                        call RadioPowerControl.start();
                        call Leds.led0On();
                }
                call ACK.requestAck(&dataPkt);
                /* @brief Sending Data via radio/phy interface */
                if(call PhySend.send(parentId[TOS_NODE_ID],
            &dataPkt,sizeof(data_t)) == SUCCESS){
                        atomic phyLock = true;
                        post taskPrint(data);
}}<%}
```

Listing 1.4. nesC source code

```
//necC code generated for slotter service
event void Slotter.fire(uint8_t slot)
{
if(slot == sleep){
    currentMacState == "SLEEP";
        if(!radioOff)
        {       printfz1("Calling radio sleep");
                call RadioPowerControl.stop();
                call Leds.led0Off();
}}
else  if(slot == waitData){
    currentMacState == "WAIT_DATA";
    printfz1("Waiting for Data");
    if(radioOff){
        call RadioPowerControl.start();
        call Leds.led0On();
}}
else  if(slot == sendData){
    currentMacState == "SEND_DATA";
        /* @brief Checking if the slot is the node's transmit slot  */
        data = (data_t*)call Packet.getPayload(&dataPkt,sizeof(data_t));
        data->nodeId = TOS_NODE_ID;
        data->destinationId = sinkId;
        data->dataSeqNo = dataSeqCount;
        if (phyLock == FALSE){
                if(radioOff){
                        /* @brief Switching on the Radio */
                        printfz1("Waking up");
                        call RadioPowerControl.start();
                        call Leds.led0On();
                }
                call ACK.requestAck(&dataPkt);
                /* @brief Sending Data via radio/phy interface */
                if(call PhySend.send(parentId[TOS_NODE_ID],
            &dataPkt,sizeof(data_t)) == SUCCESS){
                        atomic phyLock = true;
                        post taskPrint(data);
}}}
```

5.3 Implementation Evaluation

We evaluated the generated nesC source code on a hardware platform (Zolertia Z1 [23]) as mentioned in the nesC work flow shown in Fig. 12. Zolertia nodes are powered by an MSP430F2617 low power microcontroller with 16-bit RISC CPU operating at 16 MHz clock speed. It also packs in 92KB flash memory and 8KB RAM. The node is IEEE 802.15.4 compliant and uses a CC2420 transceiver operating at 2.4GHz with a data rate of 250 Kbps. Contrary to the simulation experiments, we performed certain link quality measurements using the hardware platform. We use a smaller topology than the one used in MiXiM shown in Fig. 14 to keep the experiment simple. We deployed these nodes in a corridor with each node placed

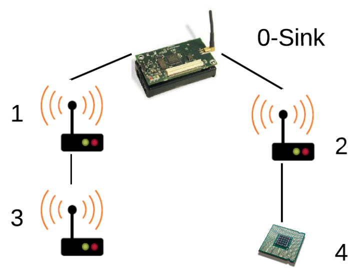

Fig. 14. The network topology used for the deployment analysis

at a distance of 5m from its parent. Thus, farthest nodes are 10m away from the sink. The corridor also had constant movement of humans. For implementation based evaluation we focused on two new metrics and mainly performed link quality assessment for presentation of basic results. Two metrics used for the assessment are Received Signal Strength Indicator (RSSI) and Link Quality Indicator (LQI). The RSSI measured for the nodes in the network is shown in Fig 15 (left) and the link quality is shown in Fig. 15 (right). RSSI indicates the strength at which the receiver receives the signal (range -45dBm to -95 dbm (lowest possible)). The LQI is a calculated assessment of the link quality based on lost bits in received packets given by TinyOS, the value ranges from 50 to 110, with values towards 110 considered to be optimal. These values were obtained from the built in TinyOS functions provided by the interface CC2420Packet. The obtained RSSI graph shows varying levels of RSSI for the nodes, which is generally based on the distance between the sender and the receiver. RSSI is also affected by interference, mainly from people moving through the corridor time to time and also, the WiFi service operating in the vicinity. LQI levels mostly lie in between 95-110 indicating the link quality is good between the nodes despite of the RSSI differences. The LQI also falls very low for one of the nodes between rounds 50 and 100 (node 4), this might be caused by continuous packet failure during a period. Further reasons for this are not provided in the basic analysis provided here since it is beyond the scope of this article. However, this is an important result since it shows the possible variation in a real environment. The LQI and RSSI values combined give a collective picture of link quality between two nodes. If both RSSI and LQI values are considered for node 4 that is affected between rounds 50 and 100 in LQI, the RSSI graph also has correspondingly low readings for those same rounds.

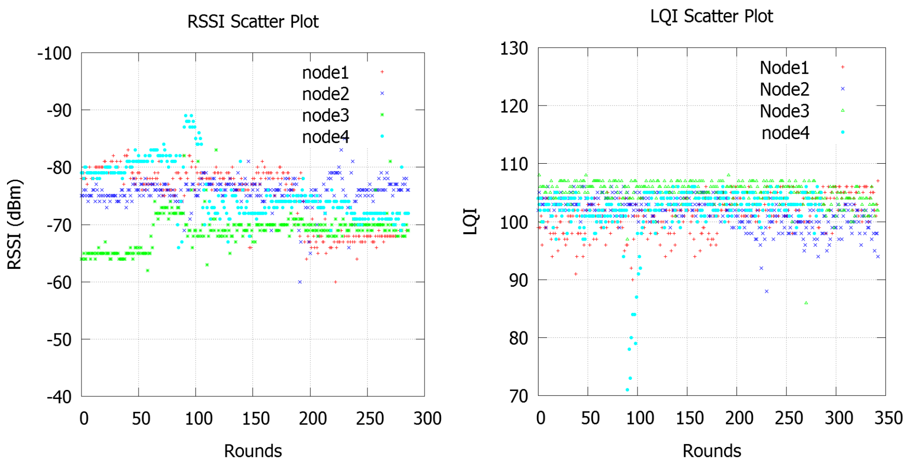

Fig. 15. Link quality based on RSSI (a) and LQI (b)

6 Conclusions and Future Work

Model-driven software engineering is a popular approach for design and development of general computing applications. We have used MDSE principles and applied it to protocol design and development in the WSN domain. In this article, we have used model-based development techniques to generate code for two different platforms: simulation and deployment, from a CPN model of the GinMAC protocol. We have used the PetriCode tool for code generation. We developed templates for MiXiM, a wireless network simulator platform, and TinyOS, an operating system for hardware platforms. We have also analyzed the generated program code to present some performance evaluations that can be obtained based on the generated code. We performed separate analyses on these two: energy consumption and lifetime on the MiXiM simulator platform, and link quality assessment using Zolertia nodes operating on TinyOS code. One important comparison between the classical methodology used for the DMAMAC protocol to the MDSE methodology used here is that the generated platform specific models are closely linked to the CPN model, thus provide a higher confidence in the generated code. The models in each step of the classical methodology on the other hand, are entirely based on requirement specification and no direct conversions are made.

In terms of future work and extension, we would like to mainly extend the code generation to specialized formal analysis tools like Uppaal or PRISM, which allows for further validation, and verification of real-time requirements of the protocols. Also, to extend the code generation to multiple platforms by exploiting their similarity with the existing code generation templates. Importantly, this applies to Castalia for simulation and Contiki OS for deployment. Castalia, is another wireless network simulation framework based on OMNeT++ and shares similarities with MiXiM. Contiki OS, an event-based operating system similar to TinyOS is an emerging operating system for low power sensor hardware, and is gaining market share rapidly. Apart from this, we would like to apply the MDSE approach to the DMAMAC protocol requirements to validate its usability.

References

1. Billington, J., Gallasch, G., Han, B.: A Coloured Petri Net Approach to Protocol Verification. In: Lect. on Concurrency and Petri Nets, pp. 210–290. Springer (2004)
2. Boonma, P., Suzuki, J.: Model-driven performance engineering for Wireless Sensor Networks with feature modeling and event calculus. In: Proceedings of the 3rd BADS Workshop. pp. 17–24 (2011)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synthesis Lectures on Software Engineering 1(1), 1–182 (2012)
4. Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., Muccini, H.: A model-driven engineering framework for architecting and analysing Wireless Sensor Networks. In: Proceedings of the 3rd SESENA Workshop. pp. 1–7 (2012)
5. Hauer, J.H.: Tkn15. 4: An ieee 802.15. 4 mac implementation for tinyos. TKN Technical Reports Series (2009)

6. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: Proceedings of the ICSE. pp. 633–642 (May 2011)
7. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN tools for modelling and validation of concurrent systems. International Journal of Software Tools for Technology Transfer 9, 213–254 (2007)
8. Köpke, A., Swigulski, M., Wessel, K., Willkomm, D., Haneveld, P.T.K., Parker, T.E.V., Visser, O.W., Lichte, H.S., Valentin, S.: Simulating wireless and mobile networks in OMNeT++ the MiXiM vision. In: Proceedings of SIMUTools (2008)
9. Kumar S., A.A., Kristensen, L.M., Øvsthus, K.: Simulation-based Evaluation of DMAMAC: A Dual-mode adaptive MAC Protocol for Process Control. In: Proceedings of the 8th SIMUTools Conference. pp. 218–227 (2015)
10. Kumar S., A.A., Øvsthus, K., Kristensen, L.M.: Towards a dual-mode adaptive MAC protocol (DMA-MAC) for feedback-based networked control systems. Procedia Computer Science 34, 505–510 (2014)
11. Kumar S., A.A., Øvsthus, K., Kristensen, L.M.: Implementation and Deployment Evaluation of the DMAMAC Protocol for Wireless Sensor Actuator Networks. In: Proceedings of the 7th ANT Conference. vol. 83, pp. 329–336 (2016)
12. Kumar S., A.A., Simonsen, K.I.F.: Towards a model-based development approach for Wireless Sensor-Actuator Network Protocols. In: Proceedings of CyPhy. pp. 35–39 (2014)
13. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: TinyOS: An operating system for sensor networks. In: Ambient Intelligence, pp. 115–148. Springer (2005)
14. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In: Proceedings of the 1st International Conference on Embedded networked sensor systems. pp. 126–137. ACM (2003)
15. Losilla, F., V Chicote, C., Alvarez, B., Iborra, A., Sanchez, P.: Wireless Sensor Network application development: An architecture-centric MDE approach. In: Software Architecture, LNCS, vol. 4758, pp. 179–194. Springer (2007)
16. Mozumdar, M.M.R., Gregoretti, F., Lavagno, L., Vanzago, L., Olivieri, S.: A framework for modeling, simulation and automatic code generation of Sensor Network Application. In: Proceedings of SECON. pp. 515–522 (2008)
17. O Donovan, T., Brown, J., Roedig, U., Sreenan, C., do O, J., Dunkels, A., Klein, A., Silva, J., Vassiliou, V., Wolf, L.: GINSENG: Performance control in Wireless Sensor Networks. In: Proceedings of the 7th SECON Conference. pp. 1–3 (2010)
18. Rodrigues, T., Batista, T., Delicato, F., Pires, P., Zomaya, A.: Model-driven approach for building efficient Wireless Sensor and Actuator Network applications. In: Proceedings of the 4th SESENA Workshop. pp. 43–48 (2013)
19. Shimizu, R., Tei, K., Fukazawa, Y., Honiden, S.: Model driven development for rapid prototyping and optimization of Wireless Sensor Network applications. In: Proceedings of the 2nd SESENA Workshop. pp. 31–36. ACM (2011)
20. Simonsen, K.I.F., Kristensen, L., Kindler, E.: Generating protocol software from cpn models annotated with pragmatics. In: Formal Methods: Foundations and Applications, LNCS, vol. 8195, pp. 227–242. Springer (2013)
21. Suriyachai, P., Brown, J., Roedig, U.: Time-critical data delivery in Wireless Sensor Networks. In: Proceedings of DCOSS. vol. 6131, pp. 216–229 (2010)
22. Vicente-Chicote, C., Losilla, F., Alvarez, B., Iborra, A., Sanchez, P.: Applying MDE to the development of flexible and reusable Wireless Sensor Networks. International Journal of Cooperative Information Systems 16(3-4), 393–412 (2007)
23. Zolertia: Z1 datasheet. `http://www.zolertia.io//`, accessed: 03-10-2015