MODELING AND EXECUTING ADAPTIVE SENSOR NETWORK APPLICATIONS WITH THE MATILDA UML VIRTUAL MACHINE

Hiroshi Wada, Pruet Boonma and Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125
email: {shu, pruet, jxs}@cs.umb.edu

Katsuya Oba
OGIS International, Inc.
San Mateo, CA 94404
email: oba@ogis-international.com

ABSTRACT
This paper proposes a model-driven development (MDD) framework to manage the complexity of application development for wireless sensor networks (WSNs). The proposed framework consists of a Unified Modeling Language (UML) profile for WSN applications and a UML virtual machine, called Matilda. The proposed UML profile abstracts the low-level details of WSNs and provides higher abstractions for application developers to graphically design and maintain their applications. Matilda is a runtime engine used to design, validate, deploy and execute WSN applications consistently at the modeling layer. This paper describes the design and implementation of the proposed MDD framework, and presents how the framework is used in WSN application development. Empirical evaluation results show that the proposed MDD framework can build efficient WSN applications.

KEY WORDS
Model-driven software development, visual modeling languages, wireless sensor networks

1 Introduction

Wireless sensor networks (WSNs) are deployed to detect events and/or collect data in physical operational environments. A variety of applications are developed with WSNs for environmental observation, structural health monitoring, human health monitoring, inventory tracking, home-/office automation and military surveillance [1, 2]. Due to the deeply-embedded pervasive nature of WSNs, they have a potential to revolutionize the way that humans understand and construct complex natural/physical systems [3].

Given this observation, WSNs have been rapidly increasing in their scale and complexity. The decrease in unit cost of sensor nodes allows WSN applications to leverage more nodes to cover larger observation areas in higher spatial resolutions. The increase in computing, storage and networking capabilities in each node allows WSN applications to implement more advanced in-network functionalities such as data filtering, node clustering, data aggregation and failure recovery. These changes in scale and complexity make WSN application development more complicated, time-consuming and error-prone. For example, even for implementing a simple node clustering mechanism (e.g., cluster member selection, cluster head election and data synchronization among members), application developers need to know many low-level mechanisms (e.g., memory management, routing, topology maintenance and signal strength sensing) and carefully control the state and behavior of each node with those mechanisms.

The complexity of WSN application development derives from two major issues: (1) a lack of adequate abstractions in application development, and (2) a lack of coherent tool chains for application development.

The first issue is a lack of adequate abstractions that application developers can leverage for the rapid implementation of WSN applications. The level of abstraction remains very low in the current practice of WSN application development. A number of WSN applications are currently implemented in nesC [4], a dialect of the C language, and deployed on the TinyOS operating system [5], which provides low-level libraries for basic functionalities such as sensor reading and node-to-node communication. nesC and TinyOS hide hardware-level details; however, they do not help developers to rapidly implement their applications. Several virtual machines and script languages are available to raise the level of abstraction in WSN application development. However, their default implementations tend not to fit application requirements. For example, in Bombilla VM [6], the default packet structure is too simple to use in many applications. Developers usually need to customize the VM (e.g., parser and packet handler) for implementing their own packet structures by using nesC and TinyOS. (Bombilla is implemented with nesC and TinyOS libraries.)

The second issue is a lack of coherent tool chains to configure and package WSN applications. In addition to programming, WSN application development involves a series of labor-intensive tasks such as the compilation and verification of program code, configuration of a simulator or sensor nodes, and deployment/injection of compiled code to nodes. Application developers need to manually work on these tasks in an-hoc manners with various tools. Those tools are often not interoperable and not well chained to improve developers' productivity. As a result, it takes considerable time through a long sequence of tasks until developers run their applications. The agility of WSN application development remains very low.

This paper proposes a new model-driven development (MDD) framework, which is intended to manage the com-

plexity of WSN application development by addressing the above two issues. The framework consists of (1) a Unified Modeling Language (UML) profile to model WSN applications, and (2) a UML virtual machine, called Matilda, to execute the models specified with the UML profile.

A UML profile extends (or specializes) the standard UML elements (e.g., class and association) in order to precisely describe domain-specific or application-specific concepts [7, 8]. The proposed UML profile abstracts away the low-level details of WSNs and provides higher abstractions for application developers (even non-programmers) to graphically design and maintain their applications. It also allows developers to understand and communicate their application designs in a visual and intuitive manner.

Matilda is a runtime execution engine (or virtual machine) for UML models [9]. It accepts a visual model defined with the proposed UML profile and directly executes the model through transforming it to executable code. Using Matilda, application developers can design, validate, deploy and execute their applications consistently at the modeling layer, without considering the low-level application details (e.g., program code and configuration files). The architecture of Matilda is designed as a pipeline of plugins. Different plugins implement different functionalities in Matilda, such as validating an input UML model against the proposed profile, generating application code, compiling/validating generated code and configuring a simulator. The pipeline architecture allows Matilda to flexibly configure its operation by replacing one plugin with another or changing an execution sequence of plugins. A plugin sequence functions as an automated tool chain to configure and package applications. This streamlines the process of WSN application development and reduces the turnaround time from application design to execution.

This paper is organized as follows. Section 2 describes the WSN application architecture that the proposed MDD framework is currently designed for. Sections 3 and 4 describe the designs of the proposed UML profile and model-to-code transformation rules, respectively. Section 5 describes the implementation of Matilda. Section 6 shows simulation results to characterize the performance of WSN applications built with the proposed MDD framework. Sections 7 and 8 conclude with some discussion on related work and future work.

2 A WSN Application Architecture

The proposed MDD framework is currently designed for a WSN application architecture, called BiSNET (Biologically-inspired architecture for Sensor NETworks) [10, 11]. This architecture is designed to address three issues in WSN applications for event detection[1]: *autonomy*–the ability to operate in unattended areas with the minimal aid from base stations and human administrators; *scalability*–the ability to scale to, for example, a large number of nodes and a large amount of data generated by nodes; and *adaptability*–the ability to adapt to dynamic conditions of WSNs (e.g., network traffic and resource availability) and dynamic conditions in nodes (e.g., sensor readings and power consumption).

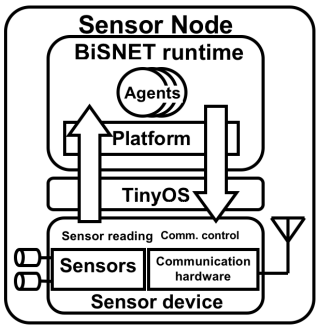


Figure 1: BiSNET Runtime Architecture

The design of BiSNET is motivated by an observation that various biological systems have developed the mechanisms to overcome the above issues [12]. For example, bees act autonomously, influenced by local environmental conditions and local interactions with other bees. A bee colony can scale to a massive number of bees because all activities of the colony are carried out without centralized control. A bee colony adapts to dynamic environmental conditions. When the amount of honey in a hive is low, many bees leave the hive to gather nectar from flowers. When the hive is full of honey, bees rest in the hive or expand the hive. Given this observation, BiSNET applies key biological mechanisms to design WSN applications [10, 11].

The BiSNET runtime consists of two software components: *agents* and *middleware platforms* (Figure 1), which are modeled after bees and flowers, respectively. Each WSN application is designed as a decentralized collection of multiple agents. This is analogous to a bee colony (application) consisting of multiple bees (agents). Agents collect sensor data (nectars) on platforms (flowers) atop nodes, and carry the sensor data to base stations, which are modeled as nests for bees. Agents perform these functionalities by autonomously invoking biological behaviors such as pheromone emission, pheromone sensing, replication and migration. A middleware platform runs atop TinyOS in each node, and hosts one or more agents. It provides runtime services that agents use to perform their functionalities and behaviors.

Inspired by biological entities (e.g., bees), agents sense their local and surrounding environment conditions, and behave adaptively according to the conditions. Each agent can perform the following behaviors.

(1) Food gathering and consumption: Biological entities strive to seek food for living. For example, bees gather nectar and digest it to produce honey. In BiSNET, agents (bees) may read sensor data (nectar) and convert it to *energy* (honey) in each duty cycle[2]. The current energy

[1] BiSNET can operate on multi-modal WSNs. A multi-modal WSN deploys multiple types of sensor nodes. Data from different types of nodes are aggregated, at base stations or through in-network processing, to provide a multi-dimensional view of collected data.

[2] The concept of energy in BiSNET does not represent the amount of

level ($E(t)$) is updated with Equation 1. S represents the absolute difference in sensor data between the current and previous duty cycles. M is metabolic rate (or energy conversion rate), which is a constant value between 0 and 1.

$$E(t) = E(t-1) + S \cdot M \quad (1)$$

(2) Pheromone emission: Agents may emit different types of pheromones (*replication* and *migration pheromones*) according to their local and surrounding environment conditions. For example, pheromone emission may occur in response to the abundance of stored energy (i.e., significant changes in their sensor readings). Different types of agents emit different types of replication pheromones, each of which contains sensor data. For example, on fluorometers, agents emit replication pheromones that contain fluorescence spectrum (fluoropheromones). On infrared sensors measuring sea surface roughness, agents emit replication pheromones that contain surface roughness data (roughness pheromones). On the other hand, agents emit migration pheromones on their local nodes when they migrate to neighboring nodes. Each pheromone has its own concentration. The concentration decays by half at each duty cycle. A pheromone disappears when its concentration becomes zero.

(3) Pheromone sensing: Agents may sense the pheromones placed on the local and neighboring nodes. This behavior is used to sense sensor data on the local and neighboring nodes.

(4) Replication: Agents may make a copy of themselves. Replication may occur in response to the abundance of energy and replication pheromones. A replicated agent is placed on the platform that its parent resides on, and it receives the half amount of the parent's energy level. Each child agent is intended to move toward a base station to report collected sensor data.

(5) Migration: Agents may move from one node to another. Migration may occur in response to energy abundance (i.e., significant changes in their sensor readings). It is used to transmit agents (sensor data) to base stations. Each agent may implement one of or a combination of the following three migration policies:

- **Directional walk:** Each agent may move to the nearest base station through the shortest path. Each base station periodically propagates a *base station pheromone* to individual nodes in the network. Its concentration decays on a hop-by-hop basis. Using base station pheromones, agents can approximate where base stations exist, and move toward the base stations by climbing pheromone gradients.
- **Chemotaxis:** Agents may follow migration pheromone traces on which many others travel. When no migration pheromones exist on neighboring nodes, agents perform directional walk.
- **Detour walk:** Each agent may go off a migration pheromone trace and follow another path to a base station when the concentration of migration pheromones is too high on the trace (i.e., when too many agents follow the same migration path). This avoids separating the network into islands. The network can be separated with the migration paths that too many agents follow, because the nodes on the paths consume more power than others and they go down earlier than others. In addition to the detour with migration pheromones, agents may avoid moving through the nodes where the concentration of replication pheromones is too high (i.e., where agents detect significant changes in their sensor readings). This distributes power consumption of agent migration over the nodes where agents detect no changes in their sensor readings, thereby avoiding network separation.

On each intermediate node toward a base station, each agent examines Equation 2 to determine which next node it migrates to.

$$WS_j = \sum_{t=1}^{3} w_t \frac{P_{t,j} - P_{t_{min}}}{P_{t_{max}} - P_{t_{min}}} \quad (2)$$

An agent calculates this weighted sum (WS_j) for each neighboring node j, and moves to a node that generates the highest weighted sum. t denotes pheromone type; P_{1j}, P_{2j} and P_{3j} represent the concentrations of base station, migration and replication pheromones on the node j. $P_{t_{max}}$ and $P_{t_{min}}$ denote the maximum and minimum concentration of P_t among neighboring nodes.

w_1 is non negative, and w_2 and w_3 can be negative. These weight values govern how agents perform the migration behavior. For example, if an agent has zero for w_2 and w_3, the agent performs directional walk by ignoring migration and replication pheromones. If an agent has a positive value for w_2, it performs chemotaxis. A negative w_2 and w_3 values allows an agent to perform detour walk.

3 A UML Profile for WSN Applications

The proposed UML profile specifies the conventions to build UML models for event detection WSN applications, and defines stereotypes and tagged-values to precisely describe computationally-complete input models[3].

In the proposed UML profile, an input model is defined as a set of UML 2.0 class diagrams, sequence diagrams and an instance diagram. A class diagram is used to define the structure of agents and sensor data that the agents carry to base stations. A sequence diagram is used to define a sequence of behaviors that an agent performs at each duty cycle. An instance diagram is used to define the topology and nodes in a WSN.

3.1 Class Diagram

Figure 2 shows an example class diagram defined with the proposed UML profile, and Figure 3 shows the defi-

physical battery in a sensor node. It is logically affects agent behaviors.

[3]"Computationally complete" means sufficiently expressive so that Matilda can interpret and execute models.

nition of stereotypes provided by the proposed profile. A class stereotyped with ≪agentType≫ represents an agent, and defines the sensor data it carries to a base station. In Figure 2, the class FluorometerAgent specifies that its instance carries fluorescenceSpectrum and timestamp to a base station.

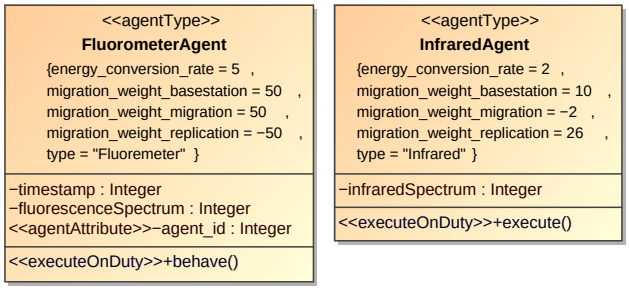


Figure 2: An Example Class Diagram

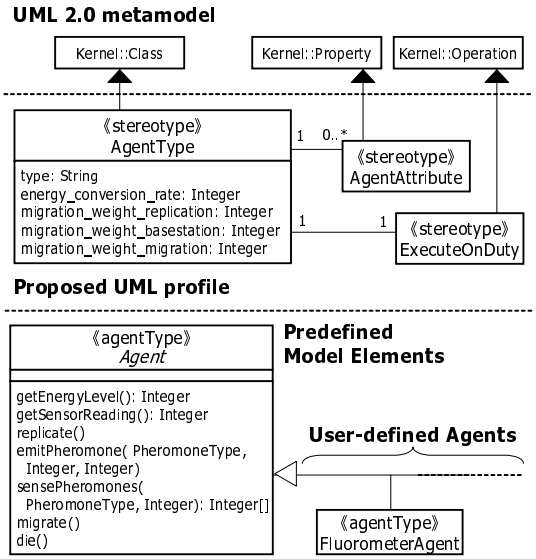


Figure 3: The Definition of the Proposed UML Profile (Class)

A series of tagged-values are used to configure each agent. The tagged-value type specifies the type of sensor device that an agent reads data from. Figure 2 shows that FluorometerAgent and InfraredAgent read data from fluorometers and infrared sensors, respectively. energy_conversion_rate specifies an energy conversion rate (or metabolic rate) used in Equation 1 (Section 2). migration_weight_basestation, migration_weight_migration and migration_weight_replication specify w_1, w_2 and w_3 in Equation 2 (Section 2).

The proposed profile provides not only stereotypes but also the Agent abstract class, which defines a set of behaviors agents invoke (see Section 3.2). Each agent is defined as an subclass of the class Agent and its parameters are configures through the use of tagged-values of ≪agentType≫.

An attribute stereotyped with ≪agentAttribute≫ represents an attribute of an agent, rather than an attribute of a message. In order to save the energy of WSNs, an agent can carry multiple messages at a time and reduce the number of message transmissions. Since normal attributes of a class, e.g., timestamp and fluorescenceSpectrum, are considered as attributes of messages, an instance of an agent may have multiple set of timestamp and fluorescenceSpectrum at a time (Figure 4). An instance of an agent, however, has only one instance (value) of an attribute stereotyped with ≪agentAttribute≫.

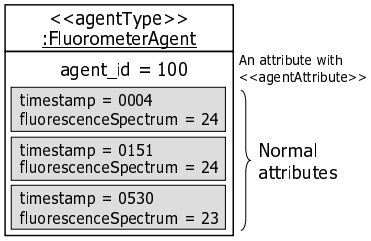


Figure 4: An Example of an Instance of an Agent

An operation stereotyped with ≪executeOnDuty≫ represents an operation that executed on each duty cycle. The behavior of an operation is defined in a sequence diagram. The proposed UML profile assumes each class with ≪agentType≫ must have one operator with ≪executeOnDuty≫.

3.2 Sequence Diagram

Figure 5 shows an example sequence diagram defined with the proposed UML profile. The purpose of sequence diagrams is to define an agent's operation by selecting bio-inspired behaviors with conditions to invoke them. A sequence diagram hides the details of implementation and allows application developers to work at a higher abstraction level, and application developers can rapidly explore the design space by developing differing versions of agent behavior.

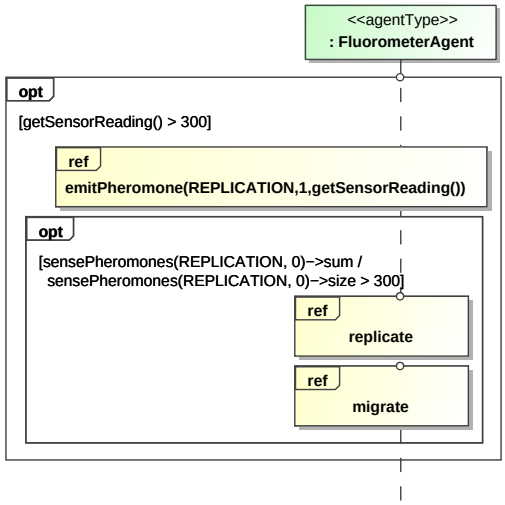


Figure 5: An Example Sequence Diagram of an Agent

As described in Section 3.1, an operation stereotyped with ≪executeOnDuty≫ executed on each duty cycle. A sequence diagram defines a control flow using InteractionOperators such as opt and alt. The opt operator designates that an enclosed fragment represents a

choice of behaviors where either the behaviors happen or nothing happens. Agent behaviors are defined as operations in the `Agent` class in Figure 2, and they can be refered as `InteractionUses`, represented as a rectangle with the label `ref`, in sequence diagrams.

The sequence diagram in Figure 5 defines an algorithm, called Gossip Filtering, to decide when an agent sends a message to a base station. An agent checks if the current sensor reading, which is obtained by `getSensorReading` operation, exceeds 300(nm). If the condition is met, an agent broadcasts the current sensor reading to one hop neighbor nodes as a replication pheromone using `emitPheromone` operation. After that, the agent checks if the average of replication pheromones from neighbors exceeds 300(nm). Each platform keeps each neighbor's replication pheromone value sent by `emitPheromone`, and `sensePheromones` operation retrieves them. The second argument specifies the number of hops to access, e.g., 0 means that the operation retrieves data from only a local node and 1 means that it retrieves data from one hop neighbors. When the second condition is met, an agent replicates itself and migrates to a base station according to Equation 2.

3.3 Instance Diagram

Figure 6 shows an example instance diagram defined with the proposed UML profile, and Figure 7 shows the definition of stereotypes which can be used in instance diagrams.

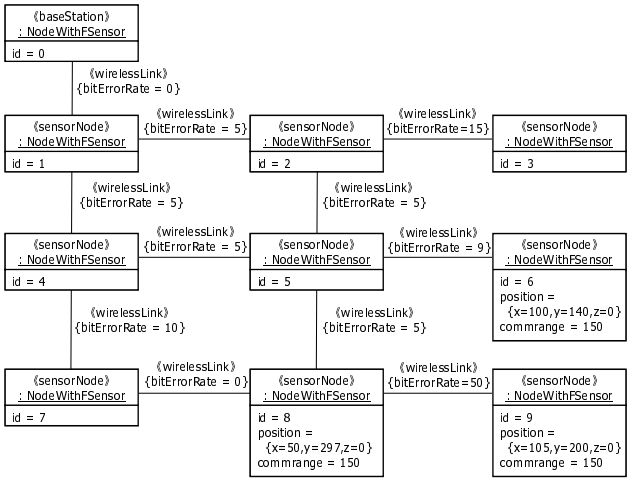


Figure 6: An Example Instance Diagram

Each deployed sensor node is represented as an instance of a class stereotyped with ≪`sensorNode`≫. The ≪`SensorNode`≫ stereotype has the `type` tagged-value representing the type of sensor node (mote), e.g., Mica2, MicaZ and Telos, and the `sensingRange` tagged-value representing a sensing range of a sensor. The proposed UML profile defines not only stereotypes but also the `Platform` abstract class, which defines a set of operations platforms can invoke. Each node is defined as a subclass of the class `Platform` as the class `NodeWithFSensor` in Figure 7.

A class with ≪`sensorNode`≫ can have one operation with ≪`executeOnDuty`≫, which is executed on each duty

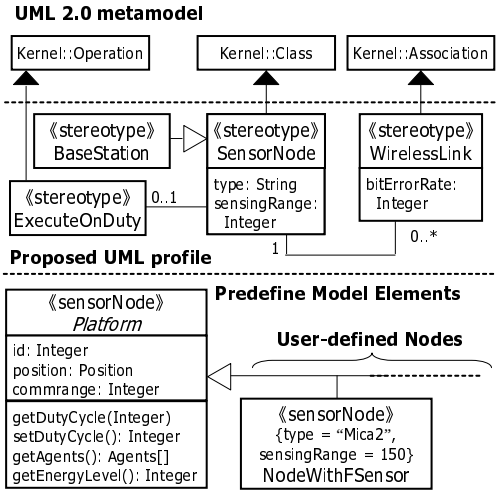


Figure 7: The Definition of the Proposed UML Profile (Instance)

cycle. It allows application developers to define platform's behavior. For example, depending on the energy level of a platform, a node changes its duty cycle (the more energy, the shorter the cycle becomes) as illustrated in Figure 8.

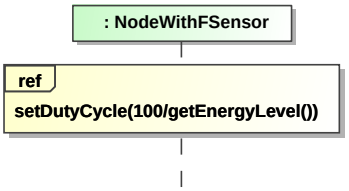


Figure 8: An Example Behavior of a Platform

As illustrated in Figure 6, a network topology is described as a set of instances of classes extending the `Platform` class. Each instance has three attributes; `id`, `position` and `commrange` represent node's ID, physical location and communication range, respectively. The attributes `position` and `commrange` are optional. For every combination of two nodes that specifies their `positions`, the quality of their communication channel is calculated from the distance between them and their `commranges`. The quality of communication channels is represented as a bit error rate, which is a value in the range (0, 100) representing the probability a bit sent by a source node will be corrupted (flipped) when a destination node receives it. For example, since nodes with `id` 6, 8 and 9 in Figure 6 (lower right) specify their `positions` and `commranges`, the qualities of their communication channels are calculated from their distances and communication ranges.

A link stereotyped with ≪`wirelessLink`≫ can be used to specify the quality of a communication channel directly. The tagged-value `bitErrorRate` represents a bit error rate. It facilitates simulating a WSN with a simple radio model, i.e., bit error rate is fixed within a communication range. Moreover, the tagged-value `bitErrorRate` can override a bit error rate calculated from the distance between two nodes. It allows simulating the situation that the quality of a communication channel cannot be calculated from the distance between nodes. For example, when there is an obstacle between two nodes, the quality of their

communication channel significantly drops. In Figure 6, since the nodes with id 8 and 9 (lower right) are connected with a link with ≪wirelessLink≫, the quality of their communication channel is specified with the tagged-value bitErrorRate regardless of the distance between them.

4 Model-to-Code Transformation

This section describes the details of the transformation rules for each UML diagram. A set of UML diagrams, i.e., class, sequence and instance diagrams, are transformed into compilable code and a configuration file for a WSN simulator. Since each agent is implemented in AgentScript, i.e., an extension of TinyScript to implement BiSNET applications, class and sequence diagrams are primarily transformed into code in AgentScript. Moreover, class diagrams are transformed into nesC code to customize Bombilla VM to support new message structures. An instance diagram is transformed into a configuration file that specifies a topology of a WSN in a simulator.

4.1 Transformation Rules for Agents

A class with ≪agentType≫ is transformed into two types of agents, i.e., stationary and migratory agents. A stationary agent stays on a node, obtains sensor readings and senses/emits pheromones. When a stationary agent sends a message to a base station, it replicates itself to create a migratory agent. A migratory agent brings sensor readings to a base station.

Listing 1 is a fragment of code generated from the class diagram in Figure 2. It is deployed on each sensor node and executed once to initialize a stationary agent when a node is activated.

Listing 1: Agent Initialization Code

```
1  agent agent = create_stationary_agent();
2  if(get_node_type() == Fluorometer) then
3   set_sensor_type(agent, Fluorometer);
4   set_energy_conversion_rate(agent, 5);
5   set_migration_weight(agent,BASETATION_PHEROMONE,50);
6   set_migration_weight(agent,MIGRATION_PHEROMONE,50);
7   set_migration_weight(agent,REPLICATION_PHEROMONE,-50);
8  else if(get_node_type() == Infrared) then
9     ...
10 end if
11 end if
```

A stationary agent is instantiated by create_stationary_agent(). Then the code checks the type of sensor that a node supports by calling get_node_type(), e.g., fluorometer sensor, infrared sensor or temperature sensor. When the type of agent specified with the tagged-value type (Figure 2) is the same as the type of sensor, a stationary agent is configured based on its tagged-values.

A sequence diagram that corresponds to an operator with ≪executeOnDuty≫ is transformed into AgentScript code, and the code is executed on each duty cycle. Listing 2 is a fragment of generated code from the sequence diagram in Figure 5.

Listing 2: Agent Behavior Code

```
1  agentlist agents = get_local_agents();
2  agent s_agent = agents[0];
3  private node_id = get_next_hop();
4  private num_of_agents = bsize(agents);
5
6  if (get_sensor_reading() > 300 ) then
7   pheromone_emission(
8      s_agent, REPLICATION, 1, get_sensor_reading() );
9   if(sum(pheromone_sensing(0))/
10       size(pheromone_sensing(0))>300) then
11    replication(s_agent);
12    for i = 1 to num_of_agents - 1
13     migration(agents[i]);
14    next i
15  end if
16 end if
```

BiSNET restricts which types of agents can perform which behaviors. For example, stationary agents can perform replication behavior, but cannot perform migration behavior. The proposed UML profile hides these restrictions, and application developers do not need to consider which agent performs which behavior. The generated code invokes behavior on appropriate agents.

In Listing 2, a stationary agent, which stored at the head of an agent list that each platform maintains, is assigned to the variable s_agent (Line 1). emit_pheromone and replication operations are called on the s_agent. migration operation is called on each migratory agent, which is stored in the agent list. The conditions to call operations are generated from conditions defined in an input sequence diagram (Figure 5).

4.2 Transformation Rules for VM Customizations

Bombilla VM is required to be customized using nesC to support new message structures corresponding to classes with ≪agentType≫. Listing 3, which is generated from the class FluorometerAgent, consists of two message structures in nesC, i.e., FluorometerAgent_Data and FluorometerAgent_Agent. As described in Section 3.1, an agent can have multiple instances of data structure consisting of normal attributes. As in Listing 3, a class with ≪agentType≫ is transformed into two data structures of which names are XXX_Agent and XXX_Data. (XXX is the name of an input class.) XXX_Agent, which defines the structure of an agent, can contain multiple XXX_Data, which defines the structure of sensor data. In this transformation, a String in UML is transformed into an array of char and an Integer is transformed into an eight bit integer (uint8_t) in nesC

Listing 3: A Definition of a Message Structure

```
1  typedef struct {
2    uint8_t timestamp;
3    uint8_t fluorescenceSpectrum;
4  } FluorometerAgent_Data;
5
6  typedef struct {
7    uint8_t agent_id;
8    FluorometerAgent_Data sensor_readings[MATE_BUF_LEN];
9  } FluorometerAgent_Agent;
```

In addition to these data structures in nesC, functions in AgentScript are also generated to handle them, e.g., functions to create an instance of new data structures, to set attribute values and to send/receive data with agents via wireless communication.

4.3 Transformation Rules for Simulator Configurations

Matilda currently uses TOSSIM, a TinyOS simulator. TOSSIM simulates a WSN based on a configuration file that specifies a set of bit error rates of communication channels between nodes. An instance diagram describing a network topology is transformed into a configuration file for TOSSIM. For example, Listing 4 is a fragment of a configuration file generated from the instance diagram in Figure 6. Each line specifies an error rate of a connection with the format `<source node ID>:<destination node ID>:<bit error rate>`. A bit error rate is in range (0, 1).

Listing 4: An Example Configuration File for TOSSIM

```
0:0:0.0
0:1:0.0
0:2:1.0
0:3:1.0
...
1:0:0.0
1:1:0.0
1:2:0.05
1:3:1.0
1:4:0.05
...
```

5 The Matilda UML Virtual Machine

This section describes the architecture of Matilda, the proposed UML virtual machine, and the implementation of each plugins for WSN event detection applications.

5.1 The Architecture of Matilda

The architecture of Matilda is designed as a pipeline of plugins. Different plugins different functionalities in Matilda, such as validating UML models, transforming UML models into compilable code, compiling and executing the code. The pipeline architecture allows Matilda to flexibly configure its structure and behavior by replacing a plugin with another one or changing the order of plugins. The pipeline architecture of Matilda is designed based on the Pipes and Filters architectural pattern [13, 14]. This pattern provides a structure of the systems that process a stream of data. The task of a system is divided into several processing steps. These steps are connected through the data flow in the system—an output data in a step becomes an input to a subsequent step. Each processing step is implemented as a filter, and filters are connected with pipes. The Pipes and Filters pattern is best suited when a system can naturally decompose its data processing task into independent steps and the requirements for the data processing task are likely to change over time. This pattern increases the reusability of filters, and allows a system to be flexible through exchanges and recombinations of filters [13].

In Matilda, each plugin works as a filter, and implements an individual step to process UML models and run an application. For example, a model loader plugin accepts a UML model in the format of XML Metadata Interchange (XMI) [15] and transforms it to an in-memory representation. A model validation plugin validates a UML model against the UML metamodel. A code generation plugin generates compilable code from the input UML model.

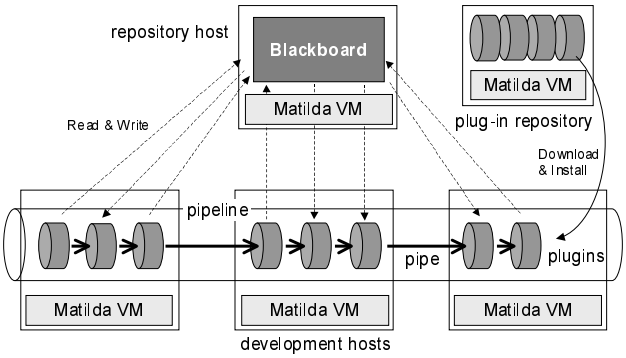


Figure 9: The Architecture of Matilda

A collection of plugins is called a pipeline (Figure 9). Each pipeline instantiates plugins and connects them with pipes based on a configuration file. The configuration file specifies plugins used in a pipeline and the execution order of plugins. Plugins are executed in a sequential or a parallel manner.

Plugins can be transparently distributed on multiple hosts in the network. Each plugin operates on a Matilda VM, which in turn runs atop of a Java VM. Plugins can be dynamically downloaded from a plugin repository and deployed in a pipeline (Figure 9). Pipelines can be configured dynamically at runtime as well as statically before executing plugins.

A common issue of the Pipes and Filters pattern is lack of robust error handling, because there is no global system state information and multiple asynchronous threads of execution exist in a system [13]. In order to overcome this issue, Matilda implements a shared repository, called *blackboard*, based on the Blackboard architectural pattern (Figure 9) [13]. This pattern is organized as a collection of independent processing units that work cooperatively on a common data structure. Each processing unit specializes to process a particular part of the overall task. It fetches data from a blackboard (shared data repository), and stores a result of its data processing to the blackboard.

In Matilda, a blackboard stores data that each plugin generates (e.g., compilable code), and makes the data available to subsequent plugins (Figure 9). It also stores the data processing log in each plugin (e.g., successful completion, errors, warnings and time stamp) in order to trace the processing status in a pipeline. In Matilda, data flow between a blackboard and plugins, and processing control flows between plugins (Figure 9).

5.2 Implementation of Plugins for WSN Applications

The pipeline shown in Figure 10 configures a set of plugins implemented for WNS applications. Plugins are categorized into two groups: *frontend* and *backend*. Frontend plugins are used to transform UML models into code and a configuration file, and backend plugins are used to customize Bombilla VM and execute WSN applications on a

TOSSIM simulator.

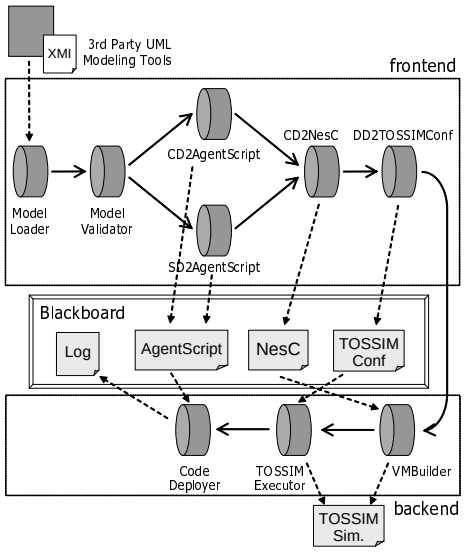


Figure 10: Typical Sequence of plugins

By leveraging the plugin `ModelLoader` plugin, Matilda accepts a UML model as an input and stores it in a blackboard (Figure 10). After that, `ModelValidator` plugin validates the input model, e.g., it checks if all classes with ≪`agentType`≫ has exactly one operation with ≪`executeOnDuty`≫. After that, `CD2AgentScript` and `SD2AgentScript` plugins transform class diagrams and sequence diagram into AgentScript code as described in Section 4.1, respectively. `SD2NesC` plugin transforms class diagrams into nesC code. `OD2TOSSIMConf` plugin transforms an instance diagram into a configuration file for a TOSSIM simulator. The `VMBuilder` plugin customizes a virtual machine and builds a TOSSIM simulator using the customized virtual machine. The `TOSSIMExecutor` plugin runs the TOSSIM simulator with a network topology specified by a TOSSIM configuration file generated by `OD2TOSSIMConf` plugin, and then `CodeDeployer` plugin deploys AgentScript code on each node in a simulator.

6 Empirical Evaluation

This simulation study emulates a WSN deployed on the sea to detect oil spills in the Dorchester Bay of Massachusetts The WSN consists of fluorometers sensors[4] deployed in an 6x7 grid topology in an area of approximately 620x720 square meters, and sensor nodes modeled after MICA2 mote with outdoor transmission range (radius) of about 150 meters, bandwidth of 38.4kbps and 128kB of program memory space (flash memory). A node at one of corners works as a base station. This study assumes that 100 barrels (approximately 3,100 gallons) of crude oil is spilled at the center of the WSN. Simulation data set is generated with an oil spill trajectory model implemented in the General NOAA Oil Modeling Environment [17]. A sensor data (fluorescent intensities) is 280nm when there is no oil, and it reaches to 318nm when there is thick oil. Since oil does not dissolve in water, the area of which fluorescent intensities is 318nm spreads during a simulation. Each sensor has a white noise that is simulated as a normal random variable with its mean of zero and standard deviation of five percent of sensor data.

This simulation study measures the differences between algorithms shown in Figure 5 (Gossip Filtering), Figure 11 (Neighborhood Filtering) and Figure 12 (Local Filtering).

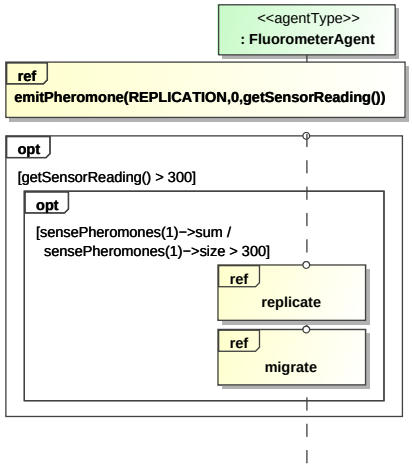


Figure 11: Neighborhood Filtering

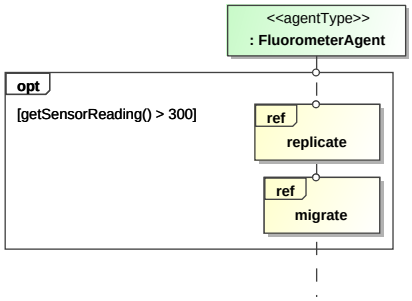
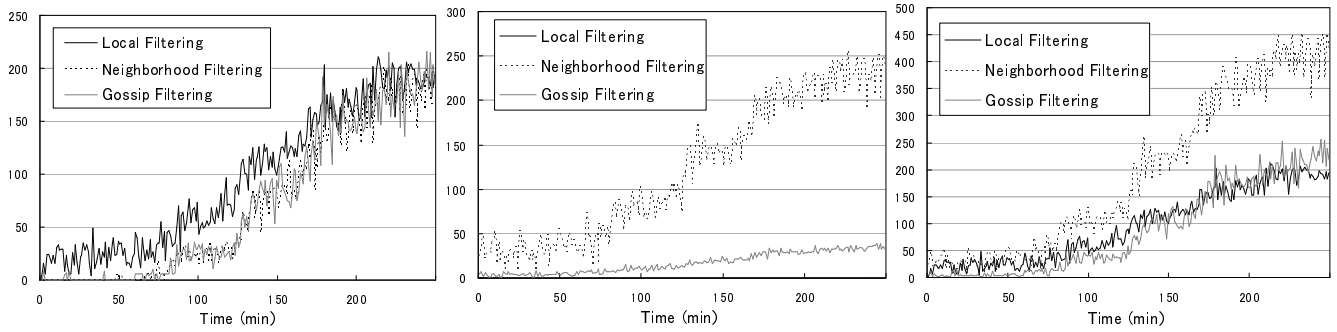


Figure 12: Local Filtering

In Neighborhood Filtering in Figure 11, each agent checks if the average of its neighbors' sensor readings exceeds 300(nm) or not to send a message to a base station. In Local Filtering in Figure 12, each agent checks if the local sensor reading exceeds 300(nm) or not.

Figure 13(a) and Figure 14 show the number of packets to transmit migratory agents from nodes to the base station and the number of false positive sensor data. Local Filtering starts sending migratory agents immediately once a sensor reading exceeds 300(nm) (Figure 13(a)) but most of them are false positive (Figure 14) since each stationary agent decides whether to send migratory agents independently. Neighborhood Filtering and Gossip Filtering use an average of sensor readings within one hop neighbors and it lowers the possibility to send false positive data. However, as shown in Figure 13(b), stationary agents send/obtain sensor readings among neighbors via replication pheromones in Neighborhood Filtering and Gossip Filtering, and it consumes much energy compare

[4]Fluorescence is a strong indication of the presence of oils. Certain compounds in oil absorb ultraviolet light, become electronically excited and fluoresce [16].



(a) Packet Transmission (b) Control Overhead (c) Total Packet Transmission

Figure 13: Packet Transmission

with Local Filtering. However, Gossip Filtering algorithm consumes less energy than Neighborhood Filtering since it uses `emitPheromone` (broadcasts) to exchange sensor data among nodes, instead of node-to-node communications using `sensePheromones`.

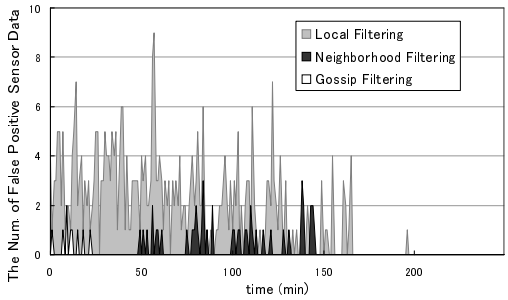


Figure 14: The Number of False Positive Sensor Data

7 Related Work

This work is extensions to the authors' prior work [9–11]. [9] investigated Matilda for a UML profile for Java command-line applications. This work retargets Matilda's application area to WSNs by proposing a UML profile for WSN applications and supporting various implementation technologies for WSNs. [10, 11] implemented BiSNET in nesC and evaluated the implementation through simulations. Based on the experience of low-level application programming, this work proposes to raise the level of abstraction in WSN application development and investigates a supporting MDD framework.

There are several work to investigate direct execution of UML models. [18] addresses the issues of validating models and generating executable code. It maintains causal connections among four meta layers in UML (M0 to M3 layers), and uses the connections to validate models and propagate changes between models. For example, the connections can be used to validate the consistency between M1 and M2 models and reflect changes in an M2 model to M1 models. Although Matilda implements model validation, it does not explicitly maintains causal connections among different meta layers. [18] does not support behavioral modeling, and it is not clear how to transform models to executable code. Matilda supports behavior modeling, and provides workable plugins to generate executable code.

Similar to Matilda, executable UML (xUML) focuses on directly executing models. In xUML, developers use class diagrams for structural modeling, and statechart diagrams and textural action languages for behavioral modeling [19–21]. Action languages implement the UML action semantics, defined as a part of the UML specification [8]. However, the UML action semantics does not provide the standard language syntax; therefore, different action languages have different syntax with different (proprietary) extensions (e.g., [22, 23]). This means that developers need to learn action language syntax every time they use different xUML tools. Also, there is no interoperability of models between different xUML tools because different xUML tools assume different subsets of the UML metamodel. Thus, an xUML tool cannot correctly interpret a model that is defined with other xUML tools. On the other hand, Matilda uses the UML metamodel and its standard extensions (profiles) for both structural and behavioral modeling. (Matilda does not require developers to use non-standard mechanisms to build and execute models.) It is more open for future extensions and integration with third party tools such as code generators and optimizers. Furthermore, Matilda inherently supports the distributed execution of plugins. No xUML tools do not address this issue.

[24] proposes a visual language to design WSN applications. The purpose of the language is to visualize language primitives in nesC, e.g., module, interface and wire, and models are transformed into nesC. The proposed profile in this paper focuses on the desing of WSN applications based on bio-inspired behaviors, rather than simply visualizing low-level language primitives. Moreover, Matilda allows running model directly. It improves the productivity of application developers. [25] proposes a simulating framework for WSNs which supports a visual language, however, it does not support code generation. Matilda also runs visual models directly, but it generates code running on actual sensor nodes. The framework proposed in [25] can be used only for simulation, but the proposed framework in this paper facilitates WSN application development.

8 Conclusion

This paper describes and empirically evaluates a new MDD framework to manage the complexity of WSN application

development. The proposed framework consists of a UML profile for WSN applications and a UML virtual machine, called Matilda. The proposed UML profile abstracts away the low-level details of WSNs and provides higher abstractions for application developers to graphically design and maintain their applications. It also allows developers to understand and communicate their application designs in a visual and intuitive manner. Matilda is a runtime execution engine (or virtual machine) to design, validate, deploy and execute applications consistently at the modeling layer, without considering the low-level application details (e.g., program code and configuration files). Matilda also serves as an automated yet customizable tool chain to configure and package WSN applications. This streamlines the process of application development and reduces the turnaround time from application design to execution. Empirical evaluation results show that the proposed MDD framework can build efficient WSN applications.

9 Acknowledgment

This work is supported in part by OGIS International, Inc. The authors would like to thank Ogidika Iloabachie, Chih-Yi Lin and Anubha Shrivastava for their contributions.

References

[1] N Xu. A Survey of Sensor Network Applications. *IEEE Communications Magazine*, 40(8), Aug. 2002.

[2] Th. Arampatzis, J. Lygeros, and S. Manesis. A Survey of Applications of Wireless Sensors and Wireless Sensor Networks. *IEEE Int'l Sym. on Intelligent Control*, June 2005.

[3] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. *ACM Int'l Conf. on Mobile Computing and Networks*, Aug. 1999.

[4] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. *ACM Conf. on Programming Language Design and Implementation*, June 2003.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, , and K. S. J. Pister. System Architecture Directions for Networked Sensors. *ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[6] P. Levis and D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. *ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[7] L. Fuentes and A. Vallecillo. An Introduction to UML Profiles. *The European journal for the Informatics Professional*, 5(2), Apr. 2004.

[8] Object Management Group. UML2.0 Super Structure Specification, Oct. 2004.

[9] H. Wada, E. M. M. Babu, A. Malinowski, J. Suzuki, and K. Oba. Design and Implementation of the Matilda Distributed UML Virtual Machine. *IASTED Int'l Conf. on Software Engineering and Applications*, Nov. 2006.

[10] P. Boonma and J. Suzuki. BiSNET: A Biologically-Inspired Middleware Architecture for Self-Managing Wireless Sensor Networks. *Elsevier Journal on Computer Networks*, 51(16), Nov. 2007.

[11] P. Boonma and J. Suzuki. An Adaptive, Scalable and Self-Healing Sensor Network Architecture for Autonomous Coastal Environmental Monitoring. *IEEE Conf. on Technologies For Homeland Security*, June 2007.

[12] T. Seeley. *The Wisdom of the Hive*. Harvard University Press, 2005.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, Aug. 1996.

[14] A. Vermeulen, G. Beged-Dov, and P. Thompson. The Pipeline Design Pattern. *ACM OOPSLA Workshop on Design Patterns for Concurrent Parallel and Distributed Object-Oriented Systems*, Oct. 1995.

[15] Object Management Group. MOF 2.0 XML Metadata Interchange, 2004.

[16] J. M. Andrews and S. H. Lieberman. Multispectral Fluorometric Sensor for Real Time in-situ Detection of Marine Petroleum Spills. In *The Oil and Hydrocarbon Spills, Modeling, Analysis and Control Conf.*, July 1998.

[17] C. Beegle-Krause. General NOAA Oil Modeling Environment (GNOME): A New Spill Trajectory Model. *Internation Oil Spill Conf.*, Mar. 2001.

[18] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The Architecture of a UML Virtual Machine. *ACM Int'l Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 2001.

[19] S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[20] C. Raistrick, P. Francis, and J. Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, Mar. 2004.

[21] M. Balcer. An Executable UML Virtual Machine. *OMG Workshop On UML for Enterprise Applications: Delivering the Promise of MDA*, June 2003.

[22] Project Technology. BridgePoint Tutorial, 2000.

[23] Kennedy Carter Ltd. The UML Action Specification Language Reference Guide, Nov. 2004.

[24] P. Volgyesi, M. Maroti, S. Dora, E. Osses, and A. Ledeczi. Software Composition and Verification for Sensor Networks. *Sci. of Computer Programming*, 56(1-2), Apr. 2005.

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