Extensible and Precise Modeling for Wireless Sensor Networks

Bahar Akbal-Delibas, Pruet Boonma and Junichi Suzuki

Department of Computer Science
University of Massachusetts Boston
Boston, MA 02125 USA
{abakbal, pruet, jxs}@cs.umb.edu

Abstract. Developing applications for wireless sensor networks (WSN) is a complicated process because of the wide variety of WSN applications and low-level implementation details. Model-Driven Engineering offers an effective solution to WSN application developers by hiding the details of lower layers and raising the level of abstraction. However, balancing between abstraction level and unambiguity is challenging issue. This paper presents *Baobab*, a metamodeling framework for designing WSN applications and generating the corresponding code, to overcome the conflict between abstraction and reusability versus unambiguity. Baobab allows users to define functional and non-functional aspects of a system separately as software models, validate them and generate code automatically.

1. Introduction

Wireless sensor networks (WSNs) are used to detect events and/or collect data in physical observation areas. They have been rapidly increasing in their scale and complexity as their application domains expand, from environment monitoring to precision agriculture, from perishable food transportation to disaster response, as just a few examples. The increase in scale and complexity make WSN application development complicated, time consuming and error prone [1].

The complexity of WSN application development derives from a lack of abstraction. A number of applications are currently implemented in nesC, a dialect of the C language, and deployed on the TinyOS operating system, which provides low-level libraries for basic functionalities such as sensor reading, packet transmission and signal strength sensing. nesC and TinyOS abstract away hardware-level details; however, they do not aid developers to rapidly implement their applications.

Model-driven development (MDD) is intended to offer a solution to this issue by hiding low-level details and raising the level of abstraction. Its high-level modeling and code generation capabilities are expected to improve productivity in WSN application development (e.g., [1, 2, 3]). However, there is a research issue in MDD, particularly in metamodeling, for WSN applications: balancing generalization and specialization in designing a metamodel for WSN applications. When metamodel designers want their metamodels to be as much generic and versatile as possible for various application domains, the metamodels can be over generalized (e.g., [2, 3]). Over generalized metamodels tend to be ambiguous and type-unsafe. Metamodel

users do not understand how to specifically use metamodel elements and often make errors that metamodel designers do not expect. Model-to-code transformers can fail due to ambiguous uses and errors that metamodel users make in their modeling work.

Another extreme in metamodeling is over specialized metamodels (e.g., [1]). Over specialized metamodels can avoid ambiguity and type unsafety; however, it lacks extensibility and versatility. Metamodel users cannot extend metamodel elements to accommodate requirements in their applications and cannot use them in the application domains that metamodel designers do not expect.

Baobab is an MDD framework that addresses this research issue for WSN applications. It provides a generic metamodel (GMM) that is versatile across different application domains. Metamodel users can use it to model both functional and non-functional aspects of their WSN applications. Baobab allows metamodel users to extend GMM for defining their own domain-specific metamodels (DSMMs) and platform-specific metamodels (PSMM). This extensibility is driven with *generics* to attain the type compatibility among GMM, DSMM and PSMM elements as well as the Object Constraint Language (OCL) [4] for avoiding metamodel users to extend GMM in unexpected ways. These two mechanisms allow application models to be type safe and unambiguous. Baobab's model-to-code transformer type-checks and validates a given application model and generates application code in nesC. It can generate most of application code, and the generated code is lightweight enough to operate on resource-limited sensor nodes such as Mica2 nodes.

2. Metamodels and Models for WSN Applications

In Baobab, metamodels are partitioned into different packages. GMM is defined in the *genericMetamodel* package.

2.1. Generic Metamodel Elements

The element Sensor of the GMM represents sensor devices that are used in WSNs. All sensor classes, representing a specific type of sensor, extend from the base class *Sensor*. The most common sensor types that can be used in a variety of applications are defined in the generic metamodel. As the names imply, each sensor detects the specific phenomenon it is prefixed by.

Nodes may send data to each other in a WSN occasionally. This can be done by packing *Data* (either some sensor reading value or a command) in a *Message*, and sending it to other nodes by a *CommunicationUnit*, which consists of a *DataTransmitter* and a *DataReceiver*. A *WirelessLink* represents the communication channel between two *CommunicationUnit*s. The sensor readings and the associated information are represented as *SensorData*. Specific classes that extend from *SensorData* will have their own attributes, as well as the shared attributes. For example, *AirTempData* has a *temperature* attribute holding the air temperature reading value. When nodes aggregate multiple *SensorData* instead of transmitting them separately, an *AggregatedData* is created. All types of *Data* can be stored in and retrieved from a *DataStorage* by *DataWriter* and *DataReader*, respectively. *EnergySource* can be used to interrogate the remaining energy level of the node.

Usage of generic types in the GMM increases the extensibility of GMM elements, as well as assuring type-safety. As an example, the *Sensor* in our generic metamodel is expected to create *SensorData*, whereas *AirTemperatureSensor* creates *AirTempData*. We defined the type of *sensorData* reference between *Sensor* and *SensorData* as a generic type that extends from *SensorData* in the generic metamodel. Thus it is feasible to associate *AirTempData* with *AirTemperatureSensor* in the GMM, and associate *BacteriaData* with *BacteriaSensor* later in the fresh-food domain metamodel.

There is a set of tasks that should be performed by a WSN node during each duty cycle. By the end of the duty cycle duration the sensor nodes will go to sleep in order to save energy. A *Timer* and a *DutyCycleManager* in the GMM manage all these series of events. At the beginning of each duty cycle *DutyCycleManager* invokes a chain of tasks to be performed, by calling the *firstTask* of the task chain defined. Each task to be performed is represented as a *Task* in the GMM. Upon completion, each *Task* will call the next *Task* defined. The tasks regarding the functional requirements of the WSN system are encapsulated in *FunctionalTasks*, whereas the tasks regarding the non-functional requirements of the WSN system are encapsulated in *NonFunctionalTasks*.

2.1.1. Functional Requirements

GMM defines several elements to express the most common functional aspects of WSNs. The functional tasks whose execution is bound to the fulfillment of a condition can be modeled by using *ConditionalFunctional* element of the GMM. This task can further be specialized into *RepetitiveTask*, which lets users to model iteration with conditions defined by the comparison of the two attributes: *repetitionNumber*, for holding the desired number of repetitions, and *repeated*, for keeping the number of repetitions completed so far. *DataReceipt* is used to define the receipt of data from another node in the network. In some cases, tasks need to be followed by a waiting period before another task can be called, which can be modeled by using *WaitingTask*. Another common functionality of WSNs, sensing phenomena, can be modeled with *SensingTask*. This element retrieves the newly created *SensorData* from the *Sensor*.

2.1.2. Non-Functional Requirements

Non-functional requirements represent the quality goals and constraints of a system. The tolerance rate of service performance, and constraints of a system are likely to change more often than the services (functional requirements) themselves in a system. Therefore, functional and non-functional aspects of a system should be modeled independent from each other. This separation not only enables developers to adapt the existing systems to new non-functional requirements easily, without annulling the whole design and creating a system from scratch, but also enables developers to reuse services in different non-functional contexts for future systems.

The non-functional requirements of a system can be modeled explicitly by means of *NonFunctionalTask* class in our GMM. The specialized non-functional tasks that are defined in the GMM are: *ClusterFormation*, for dividing the network into clusters to simplify tasks such as communication [5] and to save energy by aggregating data within the cluster; *ChangeSleepTime*, for adjusting the sleep time to minimize energy

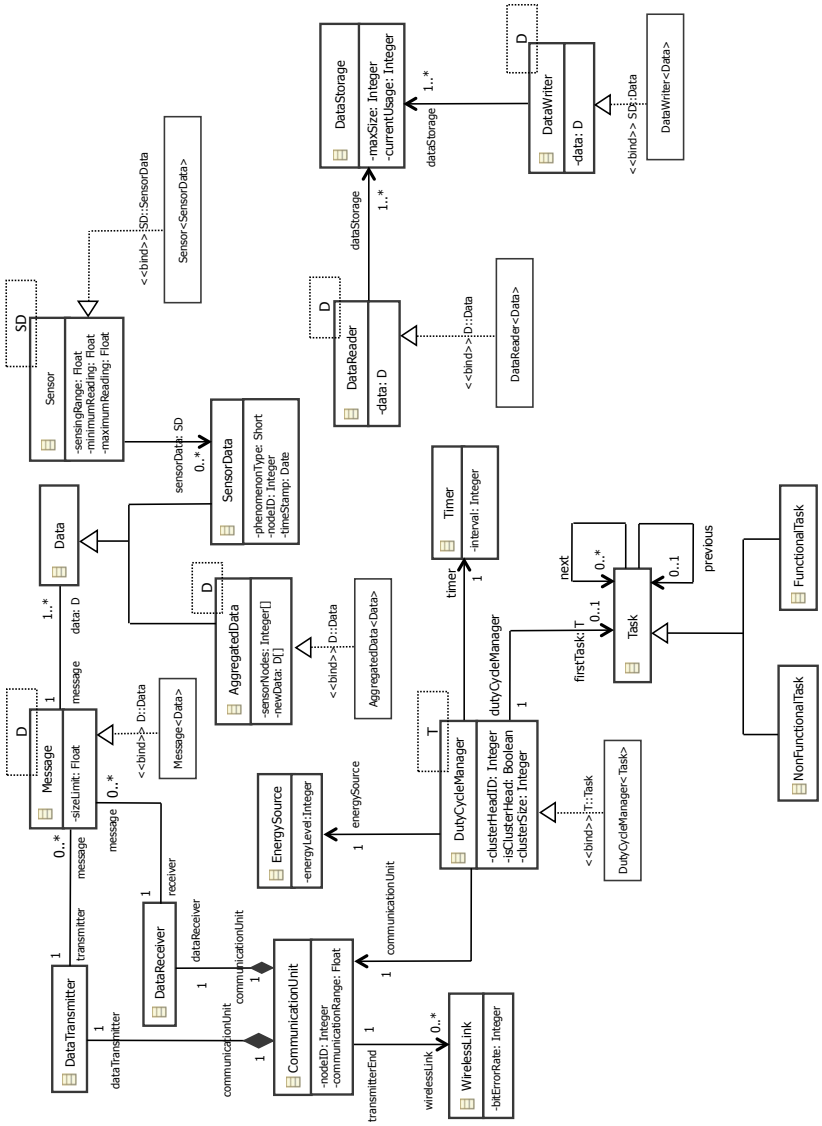


Fig. 1. Partial Generic Metamodel

consumption or to maximize data collection; *DataAggregation*, for aggregating data to be transmitted for the sake of eliminating redundancy, minimizing the number of transmissions and thus saving energy [6]; *DataTransmission*, for transmitting data with a specific policy based on the non-functional requirements; *ConditionalNonFunctional*, for specifying the activation of a *NonFunctionalTask* that is bound to fulfillment of a condition; and *ChangeCommunicationRange*, for adjusting the physical range of transmission to minimize energy consumption or to maximize data collection.

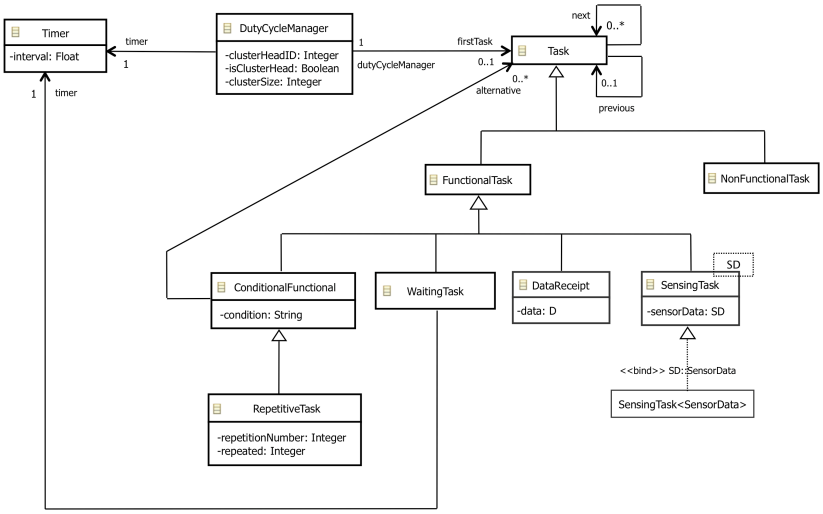


Fig. 2. Functional Aspects of the Generic Metamodel

ClusterFormation has a reference to a *ClusteringAlgorithm*, which can be one of the specialized clustering algorithms [5]. *ChangeSleepTime* and *ChangeCommunicationRange* tasks can be used to adjust the sleep time and communication range, respectively, by a given rate.

DataAggregation task has the attribute domain to specify whether the aggregation will be a *TEMPORAL* aggregation or a *SPATIAL* aggregation. The other attributes of *DataAggregation* are: *hop*, to specify how many hops away neighbors' data will be aggregated (only if *SPATIAL* aggregation domain is selected); *dutyCycleNumber*, to specify the number of duty cycles' collected data to be aggregated (only if *TEMPORAL* aggregation domain is selected); *aggregatingNodes*, the list of *nodeID*s of the neighboring nodes to aggregate data with (only if *SPATIAL* aggregation domain is selected); and *dataList*, the list of the collected data to be aggregated. The types of aggregation supported in GMM are *Average*, *Minimum*, Maximum, *Mean*, *Variance*, *MinimumAndMaximum*, *StandardDeviation*, *Suppression* (eliminating redundant data, e.g. if the temperature readings of all neighboring sensors in a region are same, only one packet containing the single sensor reading will be forwarded to the base station), and *Packaging* (combining similar data into a single message). When using *Packaging*, either of the *timeWindow* or *numberOfData* attributes should be set. Using

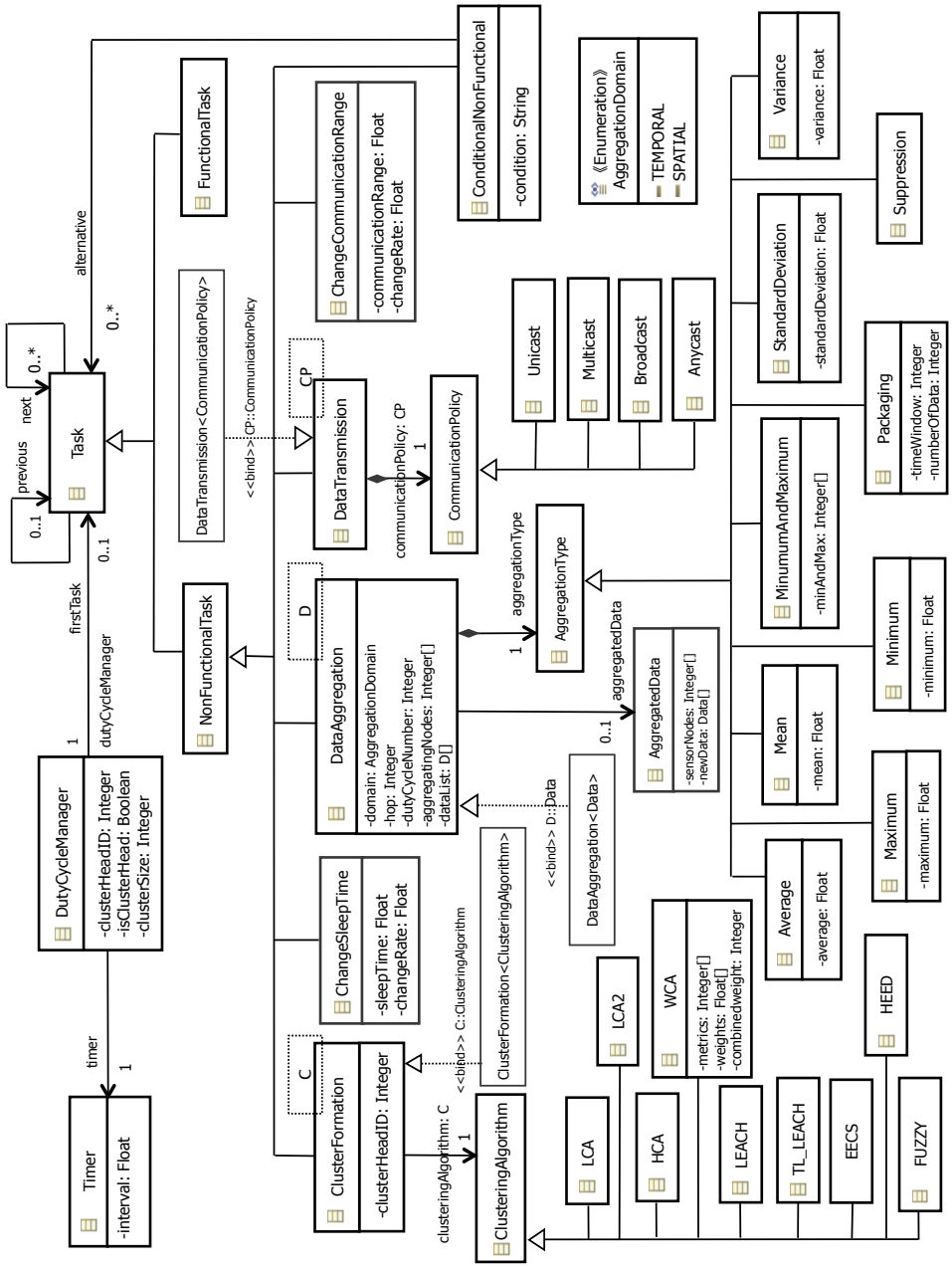


Fig. 3. Non-Functional Aspects of the Generic Metamodel

the attribute *timeWindow* denotes that exactly the same kind of data is packed together (e.g. temperature readings for the last 10 minutes), while using *numberOfData*

denotes that different kind of but related data is packed together (e.g.temperature readings and air flow readings).

For *DataTransmission* task, GMM defines four possible communication policies. *Unicast* delivers a message to a single specified node, *Broadcast* delivers a message to all nodes in the network, *Multicast* delivers a message to a group of nodes that have expressed interest in receiving the message and *Anycast* delivers a message to any one out of a group of nodes, typically the one nearest to the source.

2.2. Domain-Specific Metamodel Elements

The GMM defined in Section 2 can serve wide variety of purposes across a broad range of domains for WSNs. However, every domain may use different terminology, concepts, abstractions and constraints. Using GMM for all domains can yield to ambiguous models. The purpose of Domain-Specific Modeling is to align code and problem domain more closely. By having Domain-Specific Metamodel (DSMM) elements, Baobab helps application developers to maintain the balance between high level of abstraction and unambiguity. The GMM elements and the associated transformation rules remain the same, thus the existing models created based on the previous version of the metamodel will not be affected.

Fig. 4 shows an example DSMM for fresh food domain. In an application scenario for monitoring temperature, airflow and bacteria growth rate in a warehouse where tens to hundreds of rows of pallets of fresh meat stocked, there are several key entities and limitations that the application developers should take care of. The ambient temperatures should not be less than –1.5°C or more than +7°C throughout the cold chamber [7]. Airflow rate in the cold chamber affects the distribution of the cooled air, and setting the default air velocity to 1 m/s is ideal. The bacterial performance is measured by colony forming units (cfu/cm^2) on the surface of the meat. For the bovine meat, the acceptable range is 0 to 2 log cfu/cm^2, whereas the marginal range is between 3 to 4 log cfu/cm^2 and above 5 log cfu/cm^2 is unacceptable [8].

For creating such models, a new package of fresh food domain elements should be added to the metamodel. The user can achieve this by creating a new package named as *freshFood* under the same directory as GMM, and populating it with the necessary metamodel elements. The GMM already has *AirTemperatureSensor* and *AirTempData* so the user does not have to define them again. However, there are no sensors or specific data types defined for airflow and bacteria in the GMM. So, they are added into this new DSMM package. Possible corrective actions to be taken by the base station are: changing the airflow speed and air temperature in the cold chamber. Based on this knowledge, two new functional tasks can be defined.

2.3. Platform Specific Metamodel Elements

The GMM and the DSMM explained in the previous sections are platform-independent, in other words, they do not capture the details of the implementation language, the operating system to be deployed on, or the architecture of the application. This section explains the usage of Platform-Specific Metamodel (PSMM) elements. Separating the DSMM and PSMM results in highly re-usable models. For example, one may want to design a system by using the fresh food DSMM for mica

nodes (built upon nesC and TinyOS combination) as the target platform, using a biologically inspired architecture, and then the same domain-specific model can be re-used to design an application to work on SunSPOT (built upon Java and JVM combination), using a database-centric architecture. The metamodel elements and the transformation rules used for the fresh food domain remains the same, but the target platform specifications change.

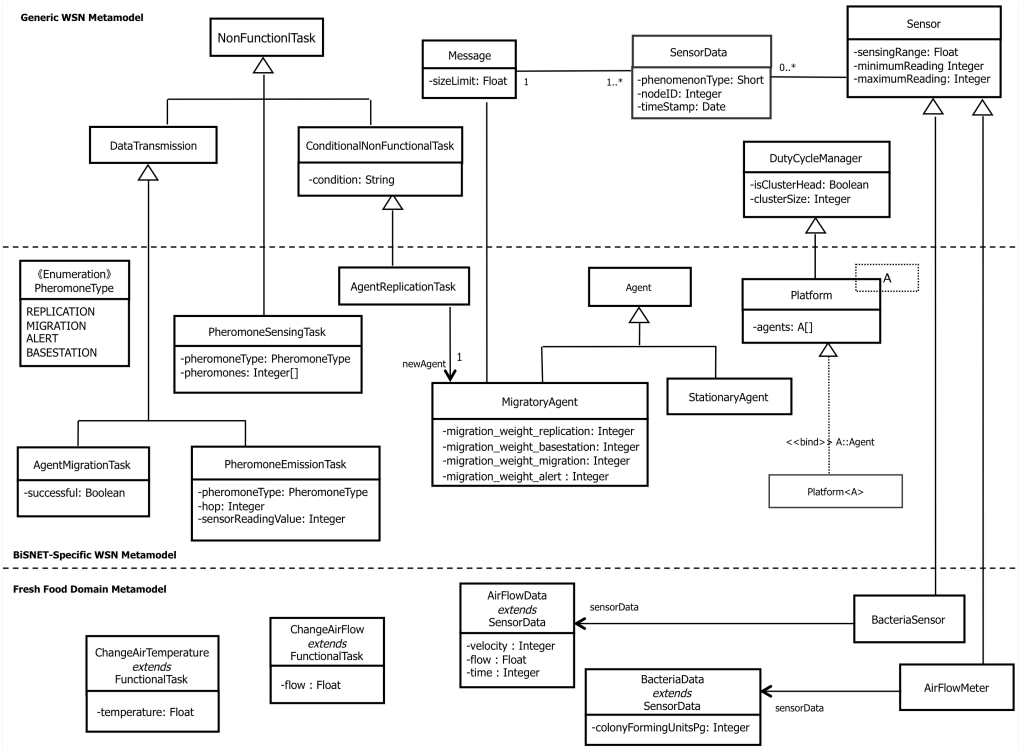


Fig. 4. Fresh Food Domain-Specific and BiSNET Platform-Specific Metamodel elements

BiSNET (Biologically-inspired architecture for Sensor NETworks) is a middleware architecture for multi-modal WSNs [9]. The two software components in BiSNET are agents and middleware platforms. Agents sense their local environments, and take actions according to sensed conditions. Upon a significant change in sensor reading an agent (a stationary agent that resides on a platform all the time) emits a pheromone to stimulate replicating itself and its neighboring agents. Each agent replicates only when enough types and concentration of pheromones become available on the local node. A replicated agent (a migratory agent) migrates toward a base station on a hop-by-hop basis to report sensor data.

PSMM elements can be added to GMM just as DSMM elements are added. A new package for each platform should be created under the same directory as GMM, and then the new package can be populated with the necessary PSMM elements extending from GMM elements. Fig. 4 depicts the resulting BiSNET PSMM.

The middleware platform and agent concepts in BiSNET are mapped to PSMM elements *Platform* and *Agent*, respectively. Since there is no entity to model software agents in GMM, Agent defined in *bisnet* package does not extend from any element of the GMM. The two types of agents, stationary agents and migratory agents, are mapped to *StationaryAgent* and *MigratoryAgent* in the *bisnet* PSMM, respectively.

2.4. Creating a Model Based on the Metamodel

There are four types of sensor nodes in the application scenario explained above: bacteria node, air temperature node, airflow node and the base station. Since Baobab considers modeling the components and functionalities of each type of node separately, four different models should be created. Fig. 5 depicts the model created for bacteria node.

3. Model Validation with OCL

Baobab allows metamodel designers to specify OCL constraints on metamodel elements so that they are extended to DSMMs and PSMMs and instantiated in models in unambiguous manner. OCL constraints can set restrictions on property values and specify dependencies between property values of an element, or different elements. Then, Baobab validates models with a given set of OCL constraints. Listing 1 shows some of the OCL constraints that are checked against the model depicted in Fig. 5.

4. Model-to-Code Transformation

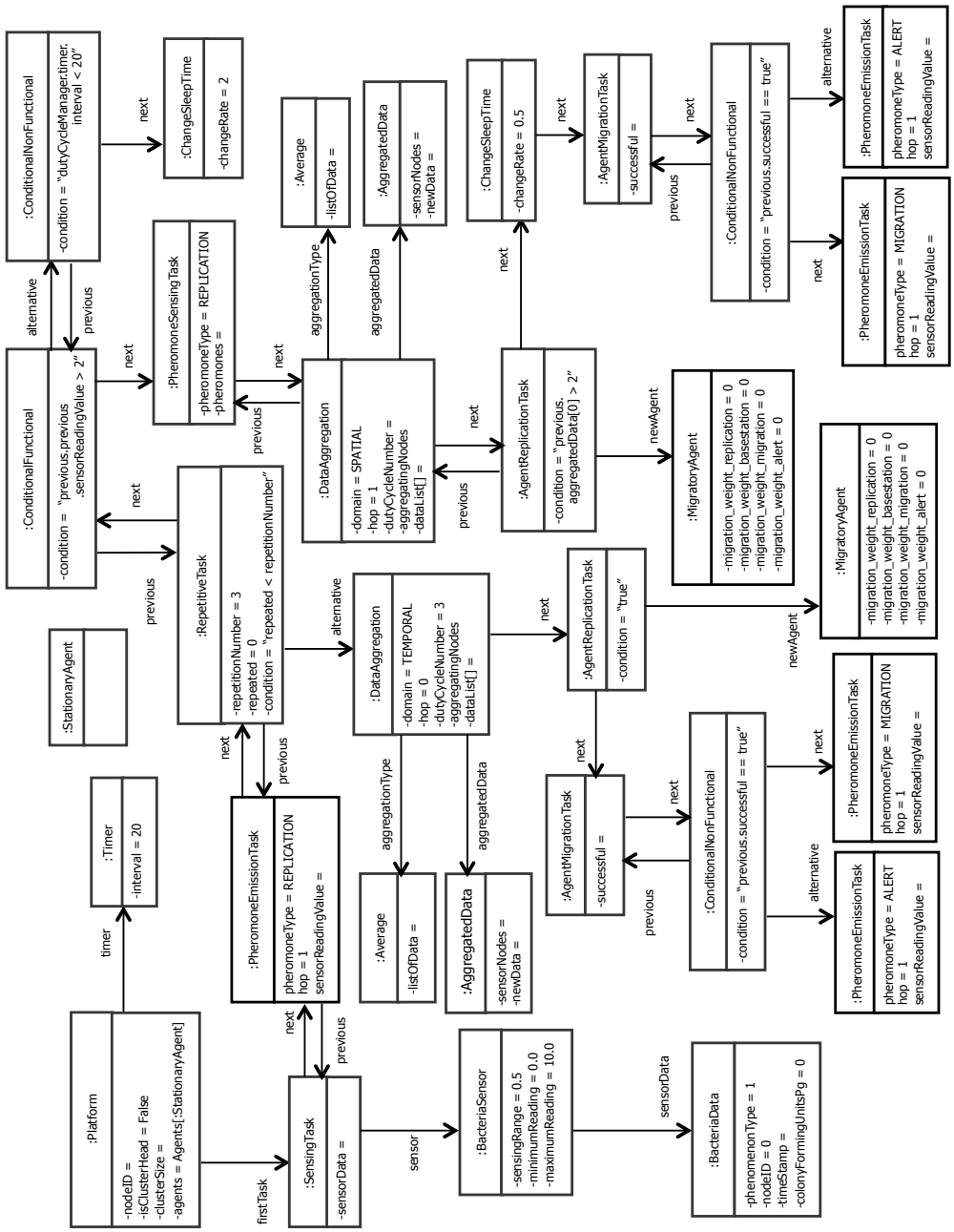
This section describes how Baobab transforms a model created with GMM, DSMMs, and PSMMs into nesC code for TinyOS. Currently, Baobab assumes that all metamodels and models are defined on Eclipse Modeling Framework[1] and uses openArchitectureware[2] to implement its model-to-code transformer.

Listing 3 is a code snippet that Babab generates from the model depicted in Fig. 5. The code performs five tasks starting with *PheromoneSensingTask*. *PheromoneSensingTask* is performed in the code by calling a BiSNET-specific function, pheromoneSensing(), with *pheromoneType* specified in Fig. 5 as a parameter. *DataAggregation* is performed by calling getAggregatedData() of the *DataAggregation* interface with relevant parameters specified in Fig. 5. getAggregatedData() takes a parameter on *aggregationType*. *AgentMigrationTask* is performed by calling migrationTask(), which is another BiSNET-specific function.

As for *AgentReplicationTask*, a conditional expression is generated as a comment in an if-statement. The actual value to be checked if it is greater than two is *aggregatedData[0]* of the previous task (*DataAggregation*). However, the generated code does not keep the previous for a task because it can be any *Task* subtype, but nesC is not an object-oriented language and it does not support polymorphism. Thus,

[1] www.eclipse.org/modeling/emf
[2] www.eclipse.org/gmt/oaw

this conditional expression is left as a comment and a programmer should replace it



with a real Boolean expression to reflect what is meant in the model.

Fig. 5. Model based on fresh food domain and BiSNET platform

Listing 1

```
-- All Data created by a Sensor should be for the same phenomenon.
  context Sensor
  inv sensorData->forAll(a1, a2 | a1 <> a2 and
       a1.phenomenonType = a2.phenomenonType)

-- AirTempSensor can only generate AirTempData.
  context AirTempSensor
  inv: sensorData->forAll(self.oclIsTypeOf(AirTempData))

-- dutyCycleNumber is used only when aggregation domain is TEMPORAL.
-- hop and aggregatingNodes should be only used when domain is SPATIAL.
  context DataAggregation
  inv: dutyCycleManager <> null implies domain = TEMPORAL
       and hop <> null implies domain = SPATIAL
       and aggregatingNodes <> null implies domain = SPATIAL

-- If aggregation domain is SPATIAL, aggregatingNodes and hop are non-full.
  context DataAggregation
  inv: domain = SPATIAL implies
              (hop <> null) xor (aggregatingNodes <> null)

-- If aggregation domain is TEMPORAL, dutyCycleNumber cannot be null.
  context DataAggregation
  inv: domain = TEMPORAL implies dutyCycleNumber <> null
```

Listing 3

```
//:PheromoneSensingTask
pheromones = pheromoneSensing(REPLICATION);
//:DataAggregation
aggregatedData = call DataAggregation.getAggregatedData(
                                                  SPATIAL, 1, 0, AVERAGE);
//:AgentReplicationTask
if(/* previous.aggregatedData[0] > 2 */) {
  int weight[4] = {0, 0, 0, 0};
  call Agent.setWeight(weight);
  agent = replicationTask(
                 aggregatedData.sensorData[0].colonyFormingUnitsPg,1);
  // :ChangeSleepTimeTask
  Timer_interval *= 0.5;
  //:AgentMigrationTask
  migrationTask(agent); }
```

5. Preliminary Evaluation

This section discusses preliminary results to evaluate Baobab. Baobab generates 1,279 lines of nesC code from the model depicted in Figure 5. It takes 544 milliseconds to generate the code. After the code is generated, there are 12 lines of code to be manually written by a programmer, which takes approximately 2 minutes. Baobab generates 99.1% of the total code; it can significantly simplify the development of WSN applications. The generated code can be deployed on the Mica2 sensor node as well as the TOSSIM simulator [10]. Table 1 shows memory footprint of the generated code on the two deployment environments. Baobab generates

lightweight nesC code that can operate on sensor nodes with severely limited resources. For example, a Mica2 node has 4 KB in RAM and 128 KB in ROM.

Table 1. Memory Footprint of a Generated WSN application

	ROM (bytes)	RAM (bytes)
Mica2	19,496	1,153
PC (TOSSIM)	72,580	179,188

6. Conclusion

This paper proposes an MDD framework, called Baobab, for WSN application development. Baobab provides a metamodel that includes the most common components and behaviors of WSN nodes, in a platform-independent way. Besides, it can be extended easily for new application domains and platforms without impairing the existing elements and rules. This metamodel also enables users to model non-functional aspects of WSN systems as well as their functional aspects. Applications can be constrained further by a set of OCL rules, and the models can be validated against these rules. The model-to-code generator creates runnable code from the input models with a little modification by the programmers.

References

1. Wada, H., Boonma, P., Suzuki, J., Oba, K.; Modeling and Executing Adaptive Sensor Network Applications with the Matilda UML Virtual Machine. Proc. of IASTED International Conference on Software Engineering and Applications (2007)
2. Vicente-Chicote, C., Losilla, F., Álvarez, B., Iborra, A., Iborra, P.: Applying MDE to the Development of Flexible and Reusable Wireless Sensor Networks. *International Journal of Cooperative Information Systems, 16(3), 393–412* (2007)
3. Sadilek, D.A.: Prototyping Domain-Specific Languages for Wireless Sensor Networks. Proc. of International Workshop on Software Language Engineering (2007)
4. The Object Management Group, Unified Modeling Language (UML) Superstructure and Infrastructure, version 2.1.2 (2007)
5. Dechene, D.J., El Jardali, A., Luccini, M., Sauer, A.: A Survey of Clustering Algorithms for Wireless Sensor Networks. Project Report (2006)
6. Krishnamachari, B., Estrin, D., Wicker, S.: The Impact of Data Aggregation in Wireless Sensor Networks. Proc. of Int'l Workshop of Distributed Event Based Systems (2002)
7. United Nations Economic Commission for Europe: UNECE Standard Bovine Meat Carcasses and Cuts. 2007 Edition, United Nations, New York and Geneva (2007)
8. Commission Regulation (EC) No 2073/2005 of 15 November 2005 on microbiological criteria for foodstuffs. Official Journal of the European Communities (2005)
9. Boonma, P., Suzuki, J.: BiSNET: A biologically-inspired middleware architecture for self-managing wireless sensor networks. *Computer Networks*, 51 (2007)
10. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. ACM Conf. on Embedded Networked Sensor Systems (2003)