EBORACUM:

DESIGN DOCUMENTATION AND TUTORIAL

Lisane Brisolara

Paulo R. Ferreira Jr.

Leandro Indrusiak

Feb. 2015.

This document presents the documentation related to the developed framework, as well as a tutorial for Eboracum users.

1. Overview

Wireless Sensor Networks (WSN) are used to monitor real environments, where different events can happen at different times and at different places. In the end-user point of view, event-triggered or reactive WSN applications can be abstractly specified as triggered events that are generated by the environment and must be handled by the system. Thus, to evaluate a WSN configuration, several application scenarios should be considered varying the load distribution in the network and the frequency of events using the appropriated probabilistic models. The definition of these scenarios is very important in the evaluation of the network efficiency and performance.

As the nodes as based on batteries, which are not rechargeable at fields, thus the energy consumption represents an important constrain in the WSN domain and determine the network life time. Simulators can help designer to analyse the behaviour of the network under a given application load, obtaining metrics are network life time and network coverage.

EBORACUM framework was developed as an extension of VisualSense and Ptolemy II, efforts from Berkeley University for simulation of wireless sensor networks (WSN) and embedded systems in general, respectively. This extension aims to simulate event-driven wireless sensor networks, allowing to analyse the network behaviour when running an event-triggered application, mainly focused in observe the discharge of the node's batteries.

2. Features

EBORACUM is an extension for Ptolemy II and VisualSense framework that provides additional modelling primitives for WSN modelling and simulation. This uses Vergil, the Ptolemy II GUI, for visual modelling and simulation. As is usual in Ptolemy II simulator, designer can build models through the instantiation of components from the libraries. To build an EBORACUM simulation model, a specific library was created that contains the components or our modelling primitives.

However, build large models manually using the GUI can be a time-consuming task. Alternatively, designer can build EBORACUM simulation models instantiating modelling primitives directly in Java. To facilitate this task, we provide also a benchmark generator that can be used like a template. Using this class, designer can define which kind of nodes they want to use, number of nodes, the node distribution, as well as some additional parameters for the network build, thus defining the platform model to be built. In the same way, designer can specify the application load model, defining which kind of events should be used, the spatial and temporal distributions for these events and computation load for each event. Both models are combined and the simulation model is automatically generated in a XML format, which can be load in the Vergil (Ptolemy II GUI) and simulated.

Moreover, our simulation generates some results in a .csv file that facilitate the data analysis and can be used to determine the service availability and network lifetime. Table 1 summarizes the data included in this report file and examples of reports can be found on the project repository.

Table 1: Data included in the simulation report

Name	Description
# sensed events	Number of events sensed, processed and whose message is reported to the sink
Total simulation time	Simulation time (should be converted for weeks or months)
# Remaining events	Number of remaining events into CPU's FIFOs of each sensor node
Remaining battery	Battery of each sensor node
Time of death	When each node die
# sent messages	Number of sent messages for each sensor node
# lost message	A lost message is defined as a message triggered by an event which is processed, but the resulting message did not arrive to the Sink

3. Simulation model: Definitions

Our simulation model is composed of two main parts: platform and application model, whose concepts and definitions are presented in this section.

3.1 Application Load Model

Our application load model is based on events that generate the computation and communication load for the WSN. An event occurs into the environment in a given spatial location, in a given time, and is related to a natural phenomenon (temperature, humidity, sound, light, presence, etc). Thus, to represent events, three aspects are relevant, which are: spatial, temporal, and functionality. Spatial is related to event's location, temporal is related to the time when this event occurs, while functionality indicates the type of sensor enabled to capture this phenomenon. Varying and combining these three aspects, one can define different events covering the most interesting natural phenomena. For example, we define that an event is static if there is no spatial variation, and atomic when it occurs in a specific time

(without temporal variation) and is not repeated. A periodic event is that recurs at intervals.

3.2 Platform Model

To support the WSN evaluation, a high-abstraction WSN platform model was defined. Basically, the platform model is composed of nodes connected by a wireless communication channel with limited range to form a given network topology. Nodes can be sensors nodes, intermediate nodes or sink nodes. Sensor nodes have a limited sensing area, which defines whether they can be affected by a specific environmental event or not. Events are only sensed by a given sensor node if the node is active at the time of the event, the event is in its sensing area and if the event type is supported by this node (i.e. the sensor is able to detect the corresponding phenomena). Following a sink-to-gateway-based architecture, a sink is an intermediate node among the gateway and the wireless sensor network itself. This sink is connected on the same radio channel as the sensor nodes.

Nodes are distributed across the area of interest and end-to-end connections among then are built, respecting the channel range and considering a way to every node to achieve the sink in a multi-hop fashion. Such information is then stored locally by each node in routing tables. If a node is inactive or dead (e.g. faulty, out of battery), connections can be dynamically rebuilt, finding another way to connect nodes avoiding inactive nodes. The proposed simulator is flexible enough to model many routing algorithms and rerouting policies.

When an event is captured by a node, its corresponding processing tasks are executed and a message is sent to the sink. To simulate this processing, our sensor nodes have a CPU, which models the execution of tasks. Currently, we assume a FIFO-scheduled CPU, but it would be trivial to support e.g. round-robin, priority preemptive or non-preemptive schedulers. As mentioned before, an event can be associated to a graph of tasks and each task can have different costs. Such costs then determine for how long a CPU is busy and how much energy it dissipates while processing a given task.

Each node has a gateway, which determines its way to achieve the sink. Thus, if the node A capture an event, it send a message to node B, its gateway, that will forward it for its gateway, until achieve the sink. Every event-detection triggers one or several message transmissions, depending on the sensor position and its distance to the sink.

Both sensor and sink nodes have a source of energy (in the current model, a non-rechargeable battery) and parameters are used to model node’s initial and current energy level. To simulate the node’s battery discharging, during the simulation, sensors and sinks will have their current battery recalculated according to their activities. This discharging process is based on CPU cycles and costs defined for each operation mode. A node can be idle, processing or communicating, where idle

means also sensing without any processing, assuming that the components responsible for sensing are always working. We assume that the reception cost for the radio is included into the idle cost. The communication cost is related only to the send message operation, thus only the node that send the message discharges its battery, but a message that take several hopes cost for every node that forward it.

4. Implementation

To support the modelling of WSN platform and application, an infra-structure named EBORACUM was implemented, which defines new modelling primitives for WSN modelling and simulation. This infra-structure is organized as a framework and can be considered as an extension of Ptolemy II and VisualSense. Basically, this infra-structure reuses the resources from Ptolemy II for visual simulation and modelling, as well as primitives from VisualSense to model wireless communication.

Extending Ptolemy and VisualSense classes, we define new modelling primitive for both layers: application and platform. As in Ptolemy II, the provided classes are implemented in Java.

Ptolemy II is based on the actor-oriented paradigm, where the system is composed of actors, coordinated by a director. The director is associated to a model of computation and coordinates the system simulation. Our simulation reuse the Ptolemy II, thus the defined primitives are used like actors. Whole defined modelling primitives are based on the *TypedAtomicActor* from Ptolemy II, which defines important attributes and methods to support the simulation according the actor-oriented paradigm. The director adopted was the *WirelessDirector*, an extension of *DEDirector* (discrete-event). Consequently, our simulation is based on the discrete-event model.

To define our platform model, some VisualSense classes were used, mainly to define ports and communication channels. *WirelessIOPort* are used to define input and output ports used to connect the nodes to the wireless communication channels, allowing the nodes send and receive messages. The class *PowerLossChannel* is used to model radio channels, whose rays are configured. In this kind of channel, the power of the signal varies according to the distance between the receiver node.

To define the sensing area of sensor nodes, wireless channels are also used. But, instead to allow communication between two nodes, this channel is used to represent the communication between sensor nodes and events. We used the *LimitedRangeChannel* to model the sensing behaviour, emulated by a sensing channel.

As in Ptolemy II, the EBORACUM classes were implemented in Java and were organized in packages. The main packages are *network and event*. The *network* package contains the classes which are used to define the platform model. This package contains a subpackage named *node*, which has classes used to model

different kind of nodes. The *event* package contains classes used to model application load.

4.1 Primitives for Platform Model

Fig. 1 illustrates a class diagram, in which are represented the classes used to define the platform model. These are part of the metamodel defined to build EBORACUM simulation models. The main component of our platform model is the Wireless Node, which is represented by the abstract class named *WirelessNode*. Nodes in our platform model can be sink or sensor nodes. Thus, the class *WirelessNode* has two subclasses, *BasicWirelessSensorNode* and *NetworkMainGateway*, to represent sensor nodes and sinks, respectively.

Usually, a sensor node must detect events and process tasks associated to the event. This behaviour is represented by the class *SimpleWSNNode,* which extends *BasicWirelessSensorNode*. The *BasicWirelessSensorNode* abstract class represents a sensor node, which has a CPU inside to process tasks associated to the detected events. The CPU is represented by the interface *SensorCPU* that specifies some methods to represent the general behaviour of a CPU. This interface has an implementation name *SimpleFIFOBasedCPU* that define the behaviour of a FIFO based CPU.

In order to allow the implementation and evaluation of different strategies for load balancing on WSN, the *BasicWirelessSensorNode* is specialized also to define the class abstract *ControledWSNNode*, which represents nodes which are controlled by another actor (the agent). To show the facilities provided by that structure, we implement a simple load balancing strategy in the class *GreedyWSNNode*.

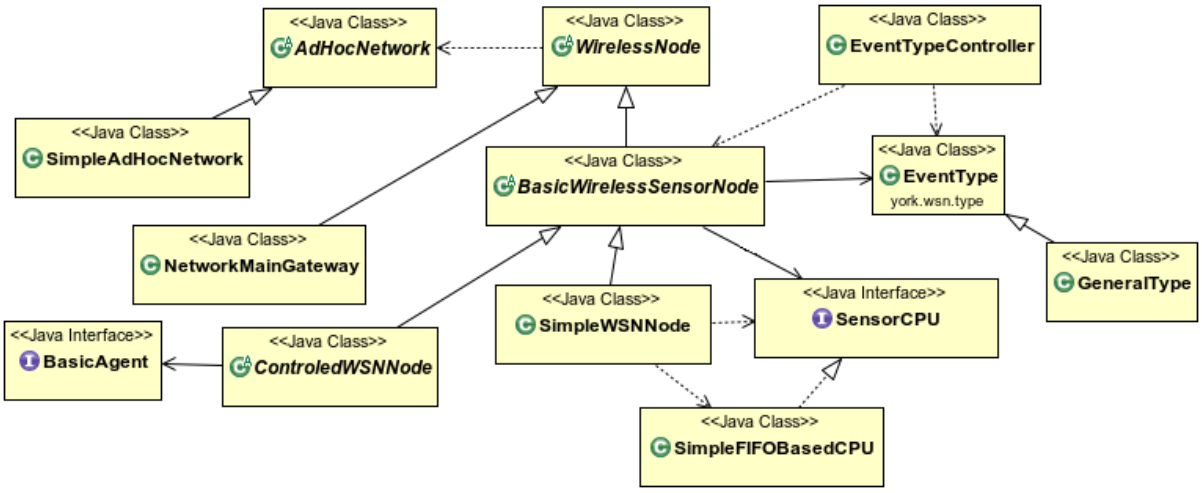

Fig. 1: Platform modelling primitives: Platform Class Hierarchy

As a framework, this platform class hierarchy is extensible and other kind of nodes can be supported, as well as other network topologies through the specialization of the defined classes.

4.2 Primitives for Application Model

Fig. 3 depicts a partial class diagram that illustrates the class hierarchy proposed to model events. Events are the basis of our application load model. On the top of this hierarchy is the *BasicEvent*, which is an abstract class that generalizes our concept of event, defining the event's attributes (e.g. position, and trigger time). *BasicEvent* is specialized by *AtomicEvent* and *PeriodicEvent* classes, which represent atomic and periodic events, respectively. An atomic event is static for nature in our hierarchy, but we propose to specialize it to represent an atomic static event whose position and trigger time are determined by a stochastic model, which is named as *StochasticStaticEvent*. Stochastic models can be any probability distribution, depicted by a histogram (2D variables) or a spectrogram (3D variable). A periodic event is an event that recurs at intervals, thus the *PeriodicEvent* class has an additional attribute named period, which represent the duration of one cycle in a repeating event.

PeriodicEvent can be used to represent events that do not change position (static) or yet events that change position. If a recurrent event does not change its position, we model it using the *StochasticPeriodicStaticEvent*, and when an event changes position, it is modelled using the *SimpleMobileEvent* or some of its subclasses or yet using *StochasticPeriodicJumperEvent*. The *StochasticPeriodicJumperEvent* can be used to model a big set of non-simultaneous static events whose spatial distribution can be defined by stochastic models, accelerating the simulation.

A mobile event is generalized by the *SimpleMobileEvent* class, which defines attributes to describe the movement parameters, such as direction, speed and time between changes. Such attributes are used to determine the new position for a mobile event, and can be also represented by probability distributions. *StochasticMobileEvent* has initial position, direction, speed, and time between direction changes determined by stochastic models, but trigger time fixed in zero, which means that the event starts on the beginning of simulation. To represent events that movement parameters and trigger time are based on stochastic models, we define the *FullyStochasticMobileEvent*. The variability of these parameters is represented by spectrograms and histograms, which define the probability distributions. For example, when representing birds monitoring, a 3D Normal function spectrogram can express the presence of young birds around the nest. A Poisson function histogram can represent for example the probability of each one move at the interval of one hour. The data represented by these curves are used to raffle a new value for the parameters. For the experiments ran in this paper, we defined also the *RandomMobileEvent*. This subclass is a simplified version of *SimpleMobileEvent*,

where the event new position is given by a small random decrease or increase on the current position.

Table 2 summarizes the modelling primitives provided to model events, which are supported by the event hierarchy (from Fig. 2). As referenced before, an event is the basis of our approach for modelling WSN application's load. Using our infra-structure is possible to represent these events, covering the three aspects: functionality, temporal and spatial. To address functionality, *BasicEvent* is associated to an event type. In the same way, the *BasicWirelessSensorNode* (from Fig. 1) is also associated to an *EventType*, thus is possible to indicate that a given node is able to sense a given event type and different kind of sensors and events can be supported. As a framework, this event hierarchy is extensible and other kind of events can be supported, modifying this hierarchy in order to include new classes.

Table 2: Modelling primitives for events

Class name (primitive name)	**Description**
AtomicEvent	Event that happens once (static)
PeriodicEvent	Event that recurs at intervals
SimpleMobileEvent	Event that changes position
RandomMobileEvent	Event that changes position using a small random decrease or increase on the current position
StochasticMobileEvent	Event has initial position, direction, speed, and time between direction changes determined by stochastic models, but trigger time fixed in zero
StochasticPeriodicStaticEvent	Recurrent event that does not change its position
FullyStochasticMobileEvent	All event parameters (movement and trigger time) are based on stochastic models
StochasticPeriodicJumperEvent	Used to model a big set of non-simultaneous static events whose spatial distribution can be defined by stochastic models

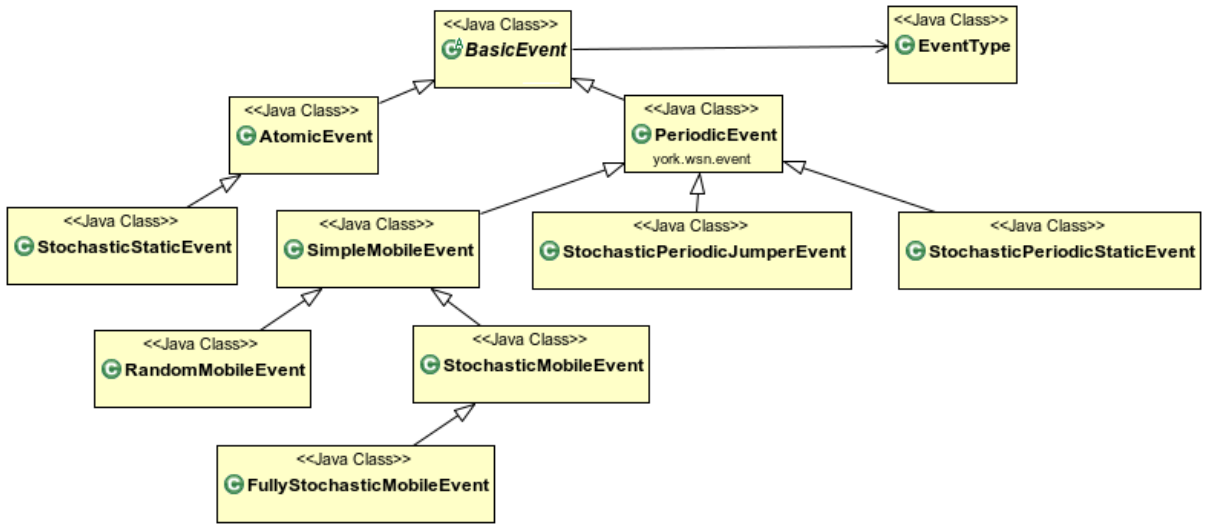

Fig. 2: Application load modelling primitives: Event Class Hierarchy

Although to evaluate efficiency, another aspect should be specified that is related to the cost to process and communicate this event. The event computation load can be yet represented as a graph of tasks, where each task can have a different cost.

Besides of the *network* and *event* packages, *simulation* package was created, which includes auxiliary classes implemented mainly for automatic generation of reports and simulation scenarios. These classes are illustrated in Fig. 3. Among these classes, *DataReporter* is actor that can be inserted into the simulation model in order to generate the report file after each simulation. The process of write the data into a file is activated in the end of simulation.

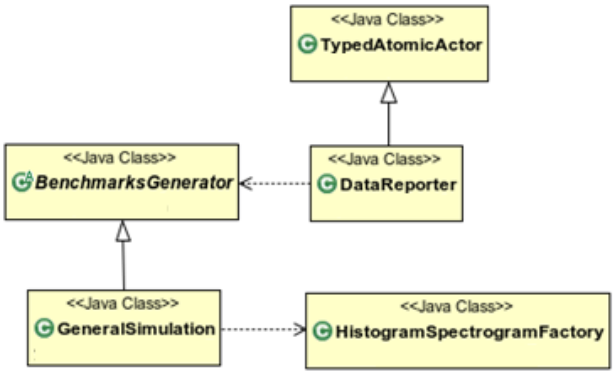

Fig. 3: Auxiliary classes

BenchmarksGenerator facilitates the building of simulation models, avoiding the manual building. This abstract class has a method responsible to generate the simulation model, according several parameters. The parameters specify which kind of nodes should be used to build the network, how many nodes should be positioned, as define the size of the area of interesting, and nodes distribution (e.g. random spatial distribution or in a grid) and the application load. This class invoke the Vergil (Ptolemy II GUI), opening the built model on Ptolemy environment. Another method from this class is the *createDataReportFile,* which defines a header for the data report, useful to identify the network configuration and simulated scenario correspondent to a given report file.

GeneralSimulation is an extension of *BenchmarksGenerator* where a simulation scenario is defined. Several parameters should be defined to build the simulation scenario, which are summarized in Table 3. More details about how use the *BenchmarksGenerator* can be found in Section 5.2.2.

Moreover, *HistogramSpectogramFactor* is a class useful to generate histograms and spectrograms for generated stochastic models used to represent stochastic events. Both *BenchmarksGenerator and HistogramSpectogramFactor* are auxiliary classes and do not part of the defined metamodel.

Table 3: Simulation Parameter: Scenario simulation

Name	Type	Description
scenarioDimensionXY	*[int,int]*	Dimension of the scenario (ex. 1000m x 1000m)
numOfNodes	*int*	Number of nodes used in the WSN solution
commCover	*double*	Cover for nodes' radio
sensorCover	*double*	Sensing Cover for all nodes
initBattery	*double*	Initial battery for the nodes
cpuCost	*double*	Energy consumption for nodes in processing state
idleCost	*double*	Energy consumption for nodes in idle state
commCost	*double*	Energy consumed for nodes to send a message
nodesRandomizeFlag	*boolean*	Used to indicate if the nodes will be random positioned in the area or not
mainGatewayCenteredFlag	*boolean*	Used to indicate if the sink is positioned in the side or centralized of the area
rebuildNetworkWhenGatewayDies	*boolean*	Used to indicate if the network should be rebuilt when a node die, finding alternative ways to achieve the sink node
synchronizedRealTime	*boolean*	Used to indicate if a simulation based on wall-clock time should be used (synchronized to real-time mode), instead of DE model

5. Instructions

This section provides some instructions about Eboracum installation and use. Short using instructions can be found also on **readme.txt**.

5.1 How to install and configure

Installing (normal users - GUI):

Download and install PtolemyII/VisualSense version 8.0.1, following the link http://ptolemy.eecs.berkeley.edu/visualsense/

To use this package you must place the file EboracumActors.xml on folder ptII8.0.1/ptolemy/actor/lib and include a tag on file ptII*.*.*/ptolemy/configs/basicActorLibrary.xml,

like: <input source="ptolemy/actor/lib/EboracumActors.xml"/>

Installing (developers - Eclipse):

Download and install PtolemyII/VisualSense version 8.0.1 as a normal user.

Configure the use with Eclipse:

http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/eclipse/windows/noSVN.htm

http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/eclipse/windows/setupClasspath.htm

http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/eclipse/windows/runPtolemyII.htm

You will need to put "-visualsense" in the running arguments of Vergil.

Copy the Eboracum package ("eboracum" folder) inside the folder ptII8.0.1/ and refresh the Eclipse packages view.

Using:

Through Vergil (using VisualSense) open file ptII8.0.1/eboracum/SampleModel.xml

5.2 How to simulate models

To simulate a WSN solution using EBOCARUM, the simulation model should be built following our modelling approach and using the provided modelling primitives.

This model should have some basic components as following: 1 Wireless Director, 1 AdhocNetwork, 1 *EventTypeController,* 2 wireless channels, the chosen number of sensor nodes. The actor *EventTypeController* is responsible to solve and verify the type of events and nodes. Beside of these required actors, *NodeRandomizer* and *Stochastic* actors can be used when the nodes should be randomly positioned into the area of interest and stochastic models will be used to define the application load.

Users can choose to build this model manually or using the benchmarks generator. Section 5.2.1 details and exemplifies how to build a simulation model manually, while in Section 5.2.2, the use of the Benchmarks generator is detailed.

5.2.1. Building a simulation model manually

Since, we use the same GUI used by Ptolemy II, users can find several tutorial and examples that shows how to use it on the Ptolemy II website (http://ptolemy.eecs.berkeley.edu/ptolemyII/).

As mentioned before, our simulation model requires the presence of some basic components. Thus, we provide a template model, or basic model, where some of these components are already included. The user should open this xml (*SampleModel.xml*) file on Ptolemy II and after that the user will insert other components to build the model (combining platform and application aspects). To build the platform model, user must choose and instantiate components from the library EBORACUM and positioning these on the work area. For example, *SimpleWSNNode* can be used to create the sensor nodes and *NetworkMainGateway* to represent the Sink. To define the application load, events should be also inserted, instantiating objects from *StochasticPeriodicStaticEvent* or *StochasticMobileEvent*. Fig. 4 presents a snapshot of simulation environment, where in the top is possible to see basic components and in the middle a simple WSN composed of three simple nodes. In the left bar is possible to observe also the library with the EBOCARUM primitives.

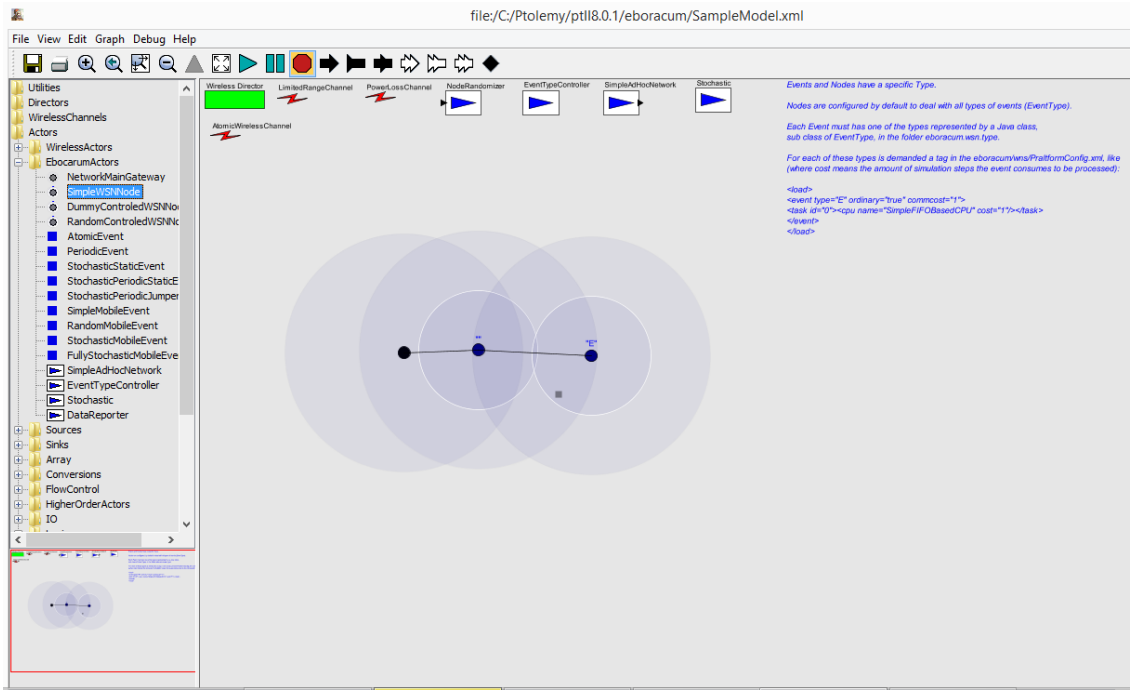

Fig. 4: Snapshot of simulation environment

5.2.2. Using the benchmark generator

The abstract class *BenchmarksGenerator* can be extended in order to create different generators for simulation scenario by implementing its method *runBenchmark*(). The code depicted in Fig. 5 represents an example of scenario generation and shows how the simulation parameters as well as platform and application models are defined through Java lines.

```
this.scenarioDimensionXY = new int[]{1000,1000};
HistogramSpectrogramFactory.newInvertNormalSpectrogram(this.scenarioDimensionXY[0]-100, this.scenarioDimensionXY[0]-
100, "spectStartPosition.csv");
this.initBattery = 5400/2;
HistogramSpectrogramFactory.newHistogram(120, "triggerTimeHist.csv");
this.commCover = 160;
this.sensorCover = 120;
int numOfNodes = 49;
this.cpuCost = 50;
this.idleCost = 0.3;
this.nodesRandomizeFlag = false;
if (!nodesRandomizeFlag) generateGridPosition(numOfNodes);
this.mainGatewayCenteredFlag = true;
this.wirelessSensorNodesType = "GeneralType";
this.wirelessNodes.put("sensor.SimpleWSNNode", numOfNodes);
this.wirelessEvents.put(new WirelessEvent("E0", 0.0018, false,"{1.0, 0.0, 0.0, 1.0}", "<task id=\"0\"><cpu
name=\"SimpleFIFOBasedCPU\" cost=\"1\"/></task>", "StochasticStaticEvent"), 20);
this.wirelessEvents.put(new WirelessEvent("E1", 0.0018, false,"{0.0, 0.0, 1.0, 1.0}", "<task id=\"0\"><cpu
name=\"SimpleFIFOBasedCPU\" cost=\"2\"/></task>", "StochasticStaticEvent"), 20);
generateEventsXML();
this.network = "SimpleAdHocNetwork";
this.rebuildNetworkWhenGatewayDies= false;
this.synchronizedRealTime = true;
generateModel(simulationIdentification);
```

Fig. 5: Code of benchmark generator implementation

In the code from Fig. 5, a simulation scenario is defined with 49 nodes distributed in a grid (7x7), and 40 events, generating the scenario illustrated on Fig. 6. In this scenario, we use two kinds of *StochasticStaticEvent* events (E0 and E1), which has different costs. This code can be found in the source code as *TestSimulation.java*, and used as a template.

The *DataReporter* actor from EBORACUM library is also automatically included in the scenario from Fig. 6. The *DataReporter* is responsible to write the file report after each simulation. This actor can be also inserted in a manually built model connected to the output port of *SimpleAdhocNetwork* (just as in Fig. 6). Table 1 summarizes the data included in this report.

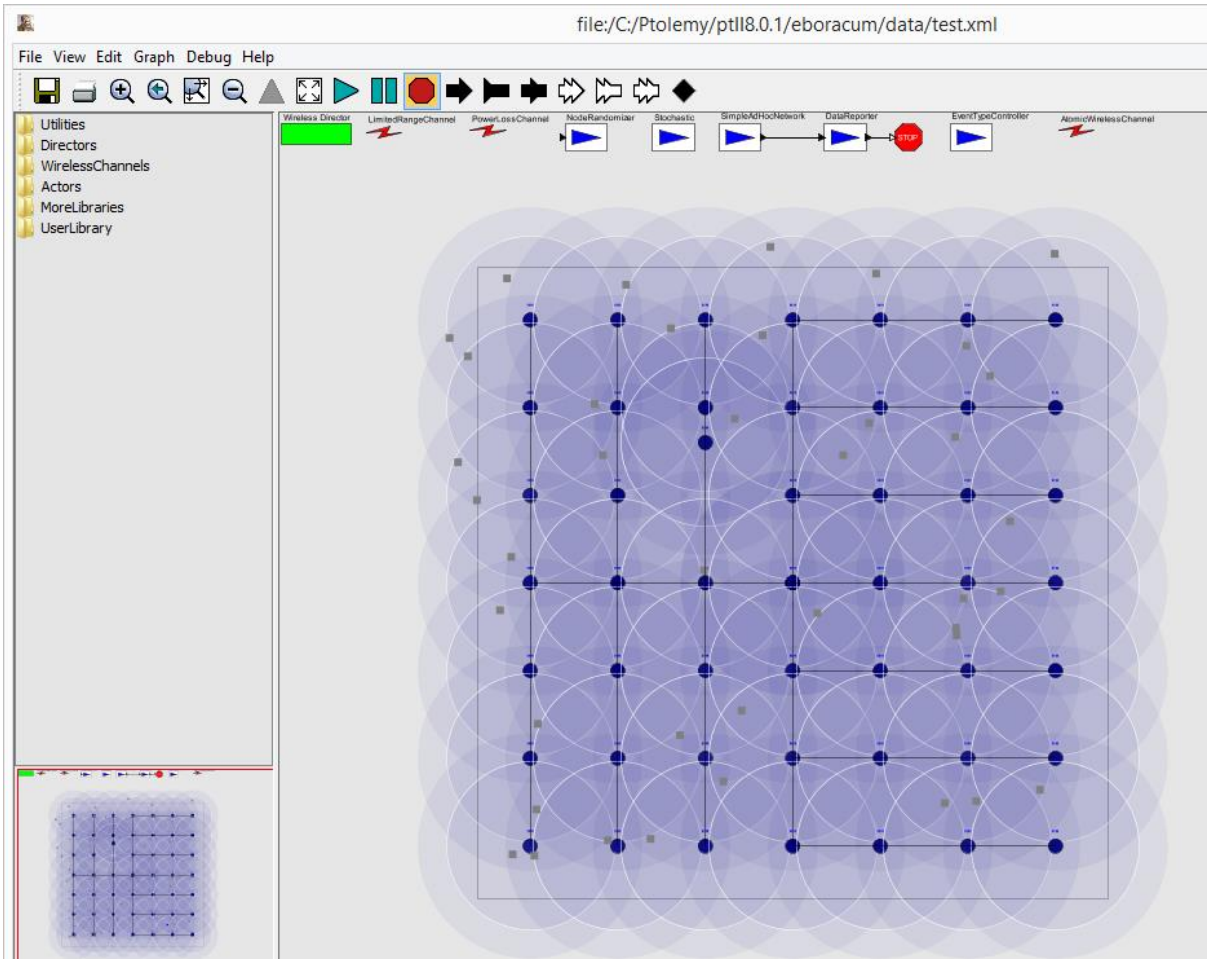

Fig. 6: Scenario automatically generated

5.2.3 How to configure costs

The evaluation of energy provided is based on several energy associated parameters related to both platform and application models. Regarding the sensor nodes, we define the *GlobalInitBattery, idleCost* and *CPUCost*, which are attributes of the actor. These attributes can be defined into Java code (as exemplified in Fig. 5)

and can be checked and modified directly into the environment visual interface, as illustrated in Fig. 7.

In the application level, for each event, we define a *commCost* and a graph of task, where each task has a cost representing the number of required CPU cycles. These application costs are specified in a XML file named *PlatformConfig.xml* and are platform-dependent. A given task can have different costs depending of the used CPU. The communication cost depends on both size of messages associated to event and radio energy consumption of the sensor node. An example of a *PlatformConfig* file can be visualized in Fig. 8. This XML was automatically generated by the benchmark generator using the code shown in Fig. 5.

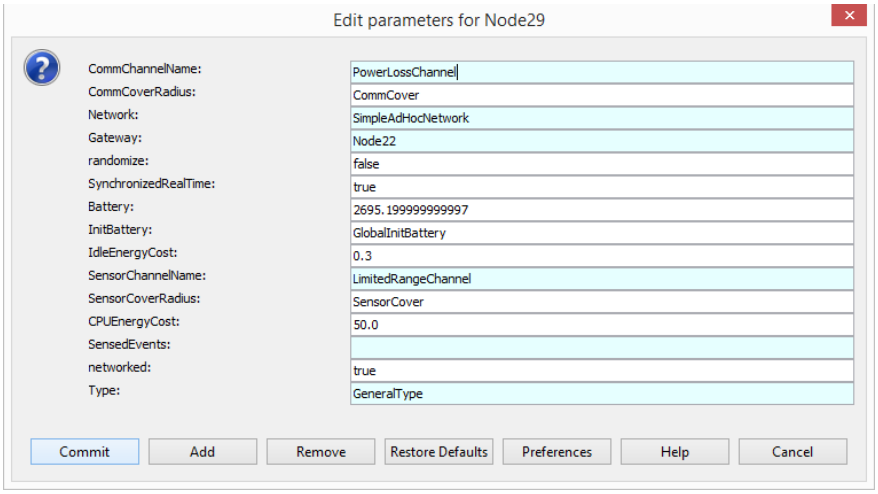

Fig. 7: Checking energy costs and battery associated to a sensor node

```
<load>
<event type="E0" ordinary="false" commcost="0.0018">
<task id="0"><cpu name="SimpleFIFOBasedCPU" cost="1"/></task>
</event>
<event type="E1" ordinary="false" commcost="0.0018">
<task id="0"><cpu name="SimpleFIFOBasedCPU" cost="2"/></task>
</event>
</load>
```

Fig. 8: Example of application/platform costs

The energy costs used in the example were defined based on to IRIS Motes datasheet, which are presented in Table 4. We assume that the battery discharge is linear and when the battery is in the half of its charge, the node stops to work. To emulate other kind of sensor nodes, the user can modify these parameters setting these for each instantiated node or specifying new values in the code of the benchmark generator.

Table 4: Energy costs and battery capacity

Energy-related parameter	**Value**
Battery capacity	5400000 mAs
Idle discharge rate	0.3 mAs
Task computation discharge rate	3.57 mAs
Discharge rate per message (3 bytes) at 30 kbps	0.0018 mAs