



# A grid-shaped cellular modeling approach for wireless sensor networks

Khaldoon Al-Zoubi<sup>1</sup> and Gabriel A Wainer<sup>2</sup>

## Abstract

WSN (Wireless Sensor Network) applications have been widely used in recent years. We introduce a new method for modeling WSN, based on the specification of the WSN using the Cell-Discrete-Event Systems Specification (DEVS) formalism: the space is partitioned into cells where each cell can be considered a sensor, an obstacle, or anything of a behavior with defined rules. This model is then converted automatically into DEVS model at runtime. We present two case studies analyzing the use of energy in WSN member nodes, which have impact on prolonging the overall network lifetime. We study to analyze energy consumption related to routing and data transmission at the node level, and topology residual energy control methods at the cluster level (i.e. group of sensors) level. The goal is to show how these spatial modeling methods can be used for building WSN models in a simple but efficient fashion.

## Keywords

Modeling and simulation, Cell-Discrete-Event Systems Specification, wireless sensor network, spatial modeling, energy modeling

## 1. Introduction

Wireless sensor networks (WSNs) are composed by a group of wireless nodes (with sensors) distributed over an area to monitor and collect data from their environment.<sup>1,2</sup> WSN nodes monitor conditions like temperature, health conditions, intrusion detection, fire detection, and military applications.<sup>1–3</sup> WSN nodes need to discover each other and form a network on demand, where nodes collect data from the environment and route/forward each other data. They need to do this while dealing with other challenges like limited power capabilities (i.e. batteries), node failures, and environmental obstacles. They have been on the rise particularly with the advent of the Internet of Things (IoT) era. In this case, sensors collect information from their environment and forward traffic toward nearby Fogs (i.e. mini clouds near users),<sup>4</sup> which might further forward data toward cloud backbone.<sup>5</sup>

To understand the complexity of such system behavior, many researchers have turned into Modeling and Simulation (M&S) to understand issues like energy consumption, routing/forwarding protocols, data collection, and many others. Accordingly, many M&S environments have been used to model and simulate WSN such as Network Simulator (NS-2),<sup>6</sup> Java Simulator (J-Sim),<sup>7</sup> Operational Network Simulator (OPNET),<sup>8</sup> OMNet++ (Modular Network Simulator),<sup>9</sup> and DEVS (Discrete-Event Systems Specification) based simulators.<sup>10</sup> In these

M&S tools (as well as in others not listed here), WSN sensors are modeled as a behavioral model (or as a piece of simulation code, usually written in a high-level programming language or a simulation tool). In most cases, the network topology is defined by interconnected by the programmer in using a scripting language, a GUI, etc. Those scripts usually define the overall topology and the interconnection between all behavioral models.

Here, we propose an alternative to the above-discussed simulators based on the definition of formal models using Cell-DEVS.<sup>10</sup> Cell-DEVS can be used to build grid-shaped cellular models, allowing allows complex spatial models to be divided into lattice of cells that only affect a close neighborhood using a discrete-event approach and explicit timing delays. Each cell represents an area that contains at most one WSN node, which affects the entire network via its neighboring nodes. A WSN model can be built with

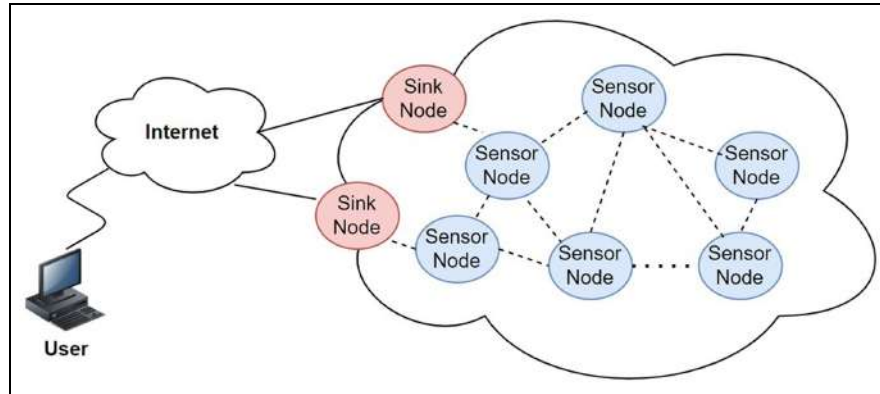
<sup>1</sup>Faculty of Computer & Information Technology, Jordan University of Science and Technology (JUST), Jordan

<sup>2</sup>Department of Systems and Computer Engineering, Carleton University, Canada

### Corresponding author:

Khaldoon Al-Zoubi, Faculty of Computer & Information Technology, Jordan University of Science and Technology (JUST), 3030 Ar-Ramtha, Irbid 22110, Jordan.

Email: ktalzoubi@just.edu.jo



**Figure 1.** Wireless sensor network (WSN) general architecture.

simple specifications that can be easily changed to try different configurations or parameters.

To demonstrate WSN modeling using spatial modeling in Cell-DEVS, we introduce two WSN models focused on energy management. Energy efficiency has direct impact on prolonging the overall network lifetime. Simply, the longer for member nodes to have enough energy to operate, the longer they stay as part of the overall network. Accordingly, energy consumption-based models have got many researchers attention. Briefly, these models attempt to save energy from a number of factors like tracking data transmission in sensors (e.g. the literature<sup>11–13</sup>) routing via shortest path (e.g. the literature<sup>14</sup>), and to avoid data collision (e.g. the literature<sup>15</sup>). In our discussed two models, we considered like other works the amount of data transmission and routing via shortest path (around obstacles). In addition, we avoid transmission via overloaded sensors to slow their energy drain. In addition, we show here how to write and develop those spatial models' specifications using Cell-DEVS in terms of their cellular rules and visualization setup. We then discuss the simulation results based on some selected visualization snapshots. We introduce the methodology and present different case studies showing the energy efficiency via routing algorithms at the node level. The model set sensors energy consumption with respect to data transmission, and it is further extended at the topology control level (i.e. across multiple clusters) showing and how the cluster head (CH) tracks energy within their clusters and route the traffic to other clusters.

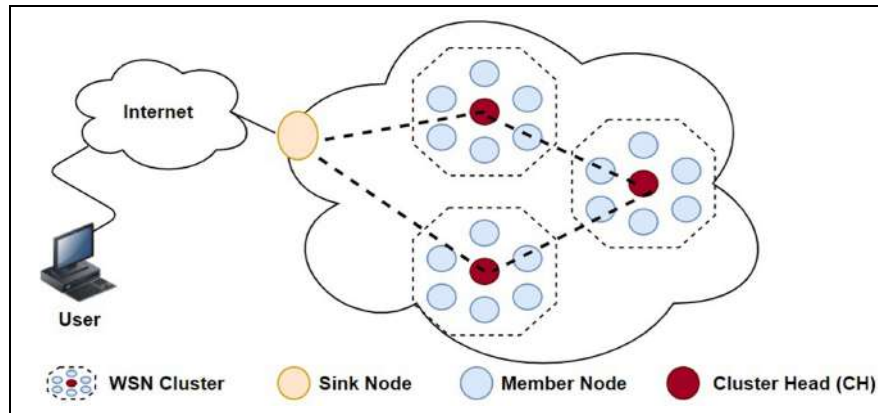
The rest of the paper is organized as follows: Section 2 provides background information and related works discussions. Section 3 provides two Cell-DEVS-based WSN energy-related models in terms of specifications and visualization results. Conclusions are presented in section 4.

## 2. Background and related works

WSN are seen as a group of wireless nodes distributed over an area to monitor and collect data from their environment and then route this data toward a centralized location to be analyzed further.<sup>1</sup> WSN nodes monitor conditions like temperature, health conditions, intrusion detection, fire detection, and military applications.<sup>1–3</sup> Each wireless node (i.e. sensor) in the network is usually equipped with a processor, memory, and power source like batteries or solar cells. In this ad hoc type of network, sensor nodes need to rely on themselves for forwarding each other data, hence without the use of typical routers, as shown in Figure 1. The data collected from the entire area can then be forwarded via the Internet to their destination for further analysis.

In this architecture, nodes act as wireless routers to their neighboring nodes where packets are continuously forwarded until they reach special nodes called sinks. Sinks push WSN packets to the external world via typical networks like the Internet; hence, sink nodes bridge the WSN with the external world. Sink nodes are usually equipped with higher processing power and more advanced technologies comparing to the regular nodes.

Because WSN can become complex, sensors usually organized in clusters where each cluster consists of group of sensors with a special node called CH, as shown in Figure 2. Each CH collects data from its local group and communicate with neighboring CH nodes to forward the data to the outside world. Multiple-cluster architecture still has sink nodes to forward the data to the external world. However, it is common for some CH nodes to serve as sink nodes as well. In this case, they collect data from their local groups and forward their local and other CH nodes data to the outside world.



**Figure 2.** WSN multiple clusters architecture.

In practice, WSN are built on the fly, particularly, when several sensors are thrown over a geographical area to collect data (e.g. from drones). Thus, energy efficiency is important requirement to extend the overall network lifetime. However, this requirement is challenging because sensor nodes have limited power capabilities. The major energy consumption activity is data forwarding as nodes not only transmit their gathered data from their environment but also route other nodes data throughout the network toward sink nodes. Therefore, the closer the nodes are to the sink nodes, the more data are transmitted and more energy is consumed. Therefore, WSN routing algorithms usually focus on optimizing the forwarding path to reduce the number of involving nodes to reduce energy consumption by the nodes involved, as energy efficiency is achieved when nodes reduce the amount of data transmitted. Data transmission is affected by obstacles in the WSN area, which also affects energy consumption of the WSN nodes.

WSN has recently been modeled and simulated by various simulation environments. Obviously, as in the case of any simulation environment, it places certain rules and constraints on how to model and simulate a specific system. Consequently, an environment technique may ease or complicate building certain models. The modeler may, for instance, need to increase the nodes size efficiently and quickly without introducing human errors. This becomes obvious in practice if a simulation tool requires the modelers to write the model in some programming language like C++ and interconnect them in scripts. To be specific, NS-2 simulator<sup>6</sup> uses (e.g. the literature<sup>16–19</sup>) a Tcl script to connect all WSN sensors where each sensor is modeled as NS-2 node. J-Sim<sup>7</sup> is like NS-2 (e.g. the literature<sup>20,21</sup>) connects sensors (called *components* and implemented in Java) using Tcl. DEVS-based<sup>10</sup> tools model (e.g. the literature<sup>22</sup>) sensors as DEVS atomic models written in programming languages like C++ or Java, and

then connect those sensors in coupled models. Similarly, OPNET<sup>8</sup> simulate (e.g. the literature<sup>11,14,15,23</sup>) WSN sensors as OPNET node models (implemented in C) interconnected as OPNET network models. Furthermore, OMNet++<sup>9</sup> simulation environment modeled (e.g. the literature<sup>13,14,24,25</sup>) WSN sensors as OMNet++ Simple module (written in C++) while the network is modeled as OMNet++ Compound module (defined in a specification language called NED). As we can see, these simulation environments define WSN sensors as behavioral units implemented in some programming language, and then interconnected with each other in groups that can also be structured hierarchically. This hierarchical structure is useful for providing scalability in the overall model structure. However, these approaches place a burden on the modeler to structure and interconnect those models particularly when the model structure grows substantially, as in the case of WSN. This issue can also be seen when the nodes size is changed in the model, which then requires the modeler to structure and reconnect the model elements. This is not a big issue if the model is small, but it can be a problem if the model contains hundreds or thousands of nodes.

Based on the above, we now provide an overview of the simulation environments that have been used to model and simulate WSN. NS-2 (network simulator)<sup>6</sup> is a discrete-event simulator with the purpose of simulation computer networks. NS-2 is Unix based and written in C++. It uses OTcl (Object-oriented Tool Command Language) to configure and setup simulation. In this case, Tcl specifications are used to build networks and interconnect its elements. Tcl specifications can then be parsed and simulated by NS-2 engine to produce simulation results files. The major parts of a Tcl specification are defining nodes and their communication links. For example, the following snippet create two nodes (sensors) and connect them together with a link of 50 MB bandwidth and 5 ms delay (i.e. \$ns is the

simulator instance variable). In this case, within Tcl script, sensors are defined as nodes and interconnected with links:

```
set node0 [$ns node]
set node1 [$ns node]
$ns duplex-link $node0 $node1 50Mb 5ms
DropTail
```

The following summarizes a few examples of NS-2-based WSN simulation.<sup>16–19</sup> Those models focused on routing protocols, data transmission, and their effect on energy consumption. The work in Shemim and Witkowski<sup>16</sup> studies various routing protocols effect on energy consumption using NS-2 simulator like Adaptive Clustering Hierarchy protocol (LEACH)<sup>17</sup> and Fuzzy-based routing protocols.<sup>18</sup> The WSN is simulated WSN sensors (i.e. modeled as NS-2 nodes) where the overall network is described in a Tcl script. The presented work in Accha and Gupta<sup>19</sup> studies MAC protocols performance and energy efficiency in WSN using NS-2. Different parameters were considered like packet delivery delay and throughput. The Tcl specification builds WSN out of 5 nodes with the following parameters: packet size is 512 bytes, Smart Medium Access Control (SMAC) duty cycle is set to 40, flow interval between nodes is set to 8 s, and so on. Finally, Awk specifications were used to extract the required information from the simulation results logs. Furthermore, the authors in work<sup>26</sup> compared Dynamic Source Routing (DSR) and AdHoc on Demand Distance Vector Routing (AODV) routing protocols in WSN via NS-2 simulation. The comparison was based on packet delivery, end-to-end delay, and throughput. The authors compared the two algorithms over two WSN constructions: one consists of 20 nodes while the other consists of 60 nodes. The overall structures were described using Tcl specifications.

OPNET<sup>8</sup> is an event-driven network simulator that offers three types of major building blocks: *network*, *node*, and *process* models. The OPNET network models offer several objects like subnets, links, and nodes within the context of a geographical area. Node models represent network elements like servers, workstations, routers, etc. Those nodes consist of different kind of objects like queues, transceivers, and processors. Finally, Process models concern with nodes internal behavior, hence consist of C code, state machines, and state variables. They mainly concern of modeling algorithms like protocols, statistics, and queuing policies.

WSN have been simulated in OPNET by representing WSN sensors as OPNET node behavioral models (implemented in C) that are grouped with OPNET network model scripts. For example, the work presented in Jun and YinSong<sup>11</sup> models WSN topology structure where line transmission monitoring is utilized by routing protocols to consume energy. The sensor is modeled as an OPNET node that keeps tracking of its distance from the sink

(edge) nodes so that traffic can be routing toward the sinks. The simulation was conducted using four clusters, hence with four CH nodes. Furthermore, the work in Zhang et al.<sup>14</sup> uses OPNET to model multi-hop WSN topology based on their geographical locations by dividing the entire area with regions centered with base stations. This was with the purpose of finding nearest location to consume energy. The model contained 400 OPNET nodes (sensors) and the geographical area was divided into four regions. Similarly, the presented work in Alsaif et al.<sup>23</sup> uses OPNET to simulate WSN behavior using ZigBee<sup>27</sup> protocol where the overall network is divided into various personal area network (PAN). In this work,<sup>23</sup> the area was divided into three PANs (i.e. modeled as OPNET network) where each PAN can contain a few mobile nodes and routers (that were modeled as OPNET nodes). Furthermore, the work in Gamal et al.<sup>15</sup> used OPNET to model and simulate the collision in WSN when sensors (modeled as OPNET nodes) wait for restricted backoff periods.

OMNet++<sup>9</sup> is a C++ discrete-event simulation framework mainly used to simulate networks and parallel/distributed systems. OMNet++ has two types of modules: Simple and Compound modules. The Simple modules are written in C++ and use the C++ class library. The Compound modules group other Simple/Compound modules (usually defined in a specification language called NED). Modules communicate with each by the means of message passing. The simulation results are then written INI files. In general, WSN basic elements like sensors were modeled as Simple Modules (in C++) while the topology of the WSN were modeled as Compound Modules.

The research in Bahbahani and Alsusa<sup>12</sup> presents an OMNet++ based WSN that uses duty cycle based on data transmission to control energy where CH is alternated between nodes to consume energy. The network was modeled as an OMNet++ compound module with 100 sensors (modeled as OMNet++ Simple module behavioral model). This behavioral model was simulated with a capacity of 3 mAh (initially charged with 5%) where sink nodes were assumed to be not of an energy constrained. Similarly, the work in Pegatoquet et al.<sup>13</sup> proposed an MAC protocol to minimize latency and energy consumption. This was done by having a centralized base station polling all nodes according to a specific interval to coordinate their data packet transmission. The OMNet++ simulation was setup as a single base station with a network with 5–50 node density. Other works like Robinson et al.<sup>24</sup> have increased the simulated OMNet++ nodes to be 500 nodes. In this work,<sup>24</sup> data were aggregated before transmission in order to consume energy. Further example, the presented work in Kodali and Malothu<sup>25</sup> proposes QoS method in WSN via controlling some parameters like bit rate and power adjustment. This system was simulated in OMNet++ (i.e. MIXIM framework) simulator environment where the model has deployed 30–100 sensors over

an area of  $500 \text{ m} \times 500 \text{ m}$ . The WSN node (sensor) was modeled with different communication layers (i.e. application, presentation, etc.). In this case, each basic block is modeled as OMNet++ Simple modules written in C++. Each group of those blocks is modeled as OMNet++ Compound modules. The overall network topology is defined in NED files (that can be viewed graphically) while exchanged messages are defined in C++ in msg files.

DEVS formalism<sup>10</sup> is currently implemented by various simulation environments. DEVS offers two types of models: the Atomic (Behavioral) model, which is usually written in some programming language like C++ or Java, and the Couple (Structural) model to interconnect other Atomic/Coupled models.

In general, WSN have been modeled in DEVS by representing sensors as DEVS atomic models while the network was represented as DEVS coupled model, while others represented the sensor as coupled model and the internal components of sensors like batteries are written as atomic models. The research in Nam and Kim<sup>22</sup> proposes a framework to model and simulate WSN systems within DEVS environments. In this framework, there are four DEVS atomic models: the sensor model (collects and forward data to intermediate sensors), BS model (analyzes reports received from sensors), GENR model (generates data randomly), and TRANSD model (handles received data from both BS and GENR models). These atomic models can then be interconnected in DEVS coupled models. In the same way, DEVS-based simulators have modeled other types of wireless networks. For example, the presented work in Tavanpour et al.<sup>28</sup> modeled mobile network using the DEVS-based CD++ simulator.<sup>10</sup> In this work, the top DEVS model (i.e. represents geographical area) in the hierarchy consists of and connects several cell-coupled models. Each cell model contains one base station atomic model with various atomic models, representing end-user equipment (e.g. mobile phone). The atomic models are implemented in C++ while the coupled models are described in textual specifications. Furthermore, one of the presented models in Tavanpour et al.<sup>28</sup> uses Cell-DEVS to model malware propagation in WSN based on the pandemic theory. This model is slightly close to what we are presenting here. However, in our case here, we focused on energy-related issues in WSN at the member node level (i.e. within a cluster) and at the topology control level (i.e. multiple clusters). Energy efficiency is essential in prolonging overall WSN lifetime as WSN sensors usually powered by limited energy source like batteries.

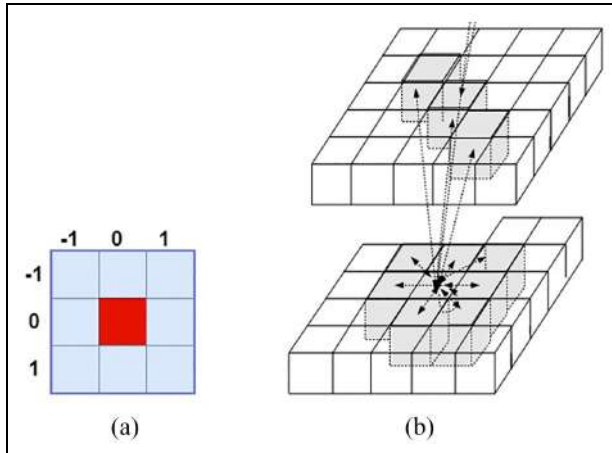
Java Simulator (J-Sim)<sup>7</sup> is a component-based simulator that implemented in Java. J-Sim is built on the top of both the INET (internetworking framework) and ACA (autonomous component architecture) simulation tools.<sup>7</sup> ACA components interact and communicate with each

other via ports. Components behavior (like the way the handle received data) is defined in what is called contracts. Components (i.e. Java classes) are connected during simulation time using Tcl specifications. J-Sim is similar to NS-2, hence used in a similar way for modeling WSN. In general, the J-Sim-based WSN simulations modeled WSN nodes as components where they can interact with each other via their ports (as defined by Tcl specifications like NS-2). For example, the work in Sobeih et al.<sup>20</sup> proposes a complete WSN framework on top of J-Sim. This work provides library for WSN elements like sensors, sink nodes, etc. These components already implement the WSN protocols in Java, allowing programmers to extend Java classes in the simulation framework to override behavior (i.e. WSN protocols). Furthermore, the work in Neves et al.<sup>21</sup> extends J-Sim simulator with WSN-specific Graphical User Interface capabilities to enhance user experience friendliness. This work helps users to avoid writing Tcl specifications manually. However, users still need to individually configure each WSN node (i.e. J-Sim component) and its connections with other WSN nodes.

There are more simulators were also built with the purpose of modeling WSN routing protocols like the graphical-based educational simulation tool for WSNs (Gbest-WSN).<sup>29</sup> This tool was based on MATLAB<sup>30</sup> and was fully dedicated to model WSN routing protocols. However, it still models WSN as nodes where the user needs to interconnect them via the GUI. In a similar way, the presented work in Gupta et al.,<sup>31</sup> which built its own simulator using IPython to simulate WSN system.

As can be seen that these above-discussed environments simulate WSN sensors as a behavioral unit that implemented in a programming language, and then interconnected with each other in groups. The above approaches are structured to provide scalability; however, it places a burden on the modeler to structure those models particularly when the model structure grows. As an alternative, we use Cell-DEVS<sup>10</sup> to model and simulate WSN. Sensors are formally modeled and then defined as a set of rules where each is placed in a cell that can communicate with a defined sensors neighborhood. This Cell-DEVS model is executed by a DEVS engine that understands the formal specifications and rules. Cell-DEVS combines DEVS formalism<sup>10</sup> with a cell space defined an  $N$ -dimension grid where each cell represents a state of the model and include computing functions. Each cell affects the overall grid of cells through a local neighborhood, which is group of cells defined in relation to a specific cell. Every time there is activity, cells states are activated and can be updated according to some function rule. This rule takes into consideration both the current cell state and the states of cells in its neighborhood. In Cell-DEVS, each cell is defined as a DEVS atomic model while a cell space is defined as DEVS coupled model, and each cell executes a local computing function to update its new state based on its current state and





**Figure 3.** Cell-DEVS neighborhood concepts: (a) a 2D near neighbors and (b) 3D complex neighborhood.

neighboring cells states. Outputs are transmitted after an explicit delay associated with the current state, as shown in Figure 3(a). Figure 3(b) shows the case when of neighboring cells in different dimensions.

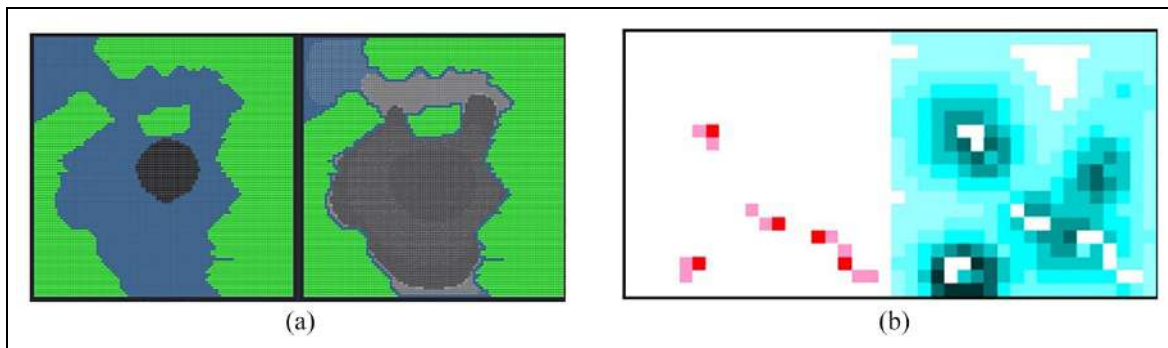
CD++<sup>10</sup> is a simulation environment that implements DEVS and Cell-DEVS formalism and can execute in different platforms, both locally and remotely.<sup>32,33</sup> It provides a complete specification language to define Cell-DEVS models. Rules (in the specification) are evaluated by CD++ in sequence until a rule condition is satisfied, which accordingly update the cell state variables and communicate information throughout the neighborhood after a specified delay. The rule format is as follows: *rule: {Post-State} {Delay} {Condition}*. This means if *Condition* is evaluated to true, then the *Post-State* value is executed, and after the specified time *Delay* expression, the output is transmitted (to the neighbors or other DEVS models). However, if *Condition* is evaluated to false, it moves to the next rule. For example, the following rule “*rule: 2 10 {(0,0) = 0 AND (-1,1) = 1}*” means that if current cell

state is 0 and neighbor cell  $(-1,1)$  state is 1, then change current cell state to 2, after a delay of 10.

Defining new models using this method is relatively simple compared with other traditional methods. The use of Cell-DEVS also enables integration with other existing models, permitting to define multi-formalism applications. An advantage of this approach is that in ad hoc models are often composed of different subcomponents (for instance, the routing algorithm and the city topology) interacting together. We can also make use of existing infrastructure, including parallel simulators and distributed environments, and a variety of visual tools. The use of a discrete-event approach (with a continuous time base) improves precision and efficiency of the models (as we only react to events in the model, and if there is no activity, the cells are not updated). This method also allows to build visualizations to track the simulation progress of CD++ simulations. For example, Figure 4(a) shows a visualization example for a particles diffusion model,<sup>10</sup> while Figure 4(b) shows different stages of a tumor-immune system simulation.<sup>10</sup>

### 3. Modeling WSN using Cell-DEVS

This section presents two different Cell-DEVS-based models that show how to define and study various WSN behaviors. The first model (section 3.1) focuses on the energy consumption problem at the node-level, hence at the level of data transmission and routing between nodes until data packets reach sink nodes. This model can be viewed as the internal behavior of clusters. Similar to other related works (discussed in section 2), sensors energy is saved via tracking data transmission in sensors (e.g. the literature<sup>11,12</sup>) and routing via shortest path (e.g. the literature<sup>13,14</sup>). In addition, we avoided transmission via overloaded sensors to slow their energy drain. The second model (section 3.2) focuses on topology control methods, which use residual energy; hence, this model is at the level of multiple clusters.



**Figure 4.** Cell-DEVS visualization examples: (a) coastal oil spill and (b) UAV search strategies.

```

[wsn_cell]
type : cell    dim : (20,20)
delay : transport    defaultDelayTime : 10
border : unwrapped
neighbors : wsn_cell(-1,-1) wsn_cell(-1,0) wsn_cell(-1,1)
           wsn_cell(0,-1) wsn_cell(0,0) wsn_cell(0,1)
           wsn_cell(1,-1) wsn_cell(1,0) wsn_cell(1,1)
localtransition : wsnBehavior
statevariables : stRoute stRoute1 stRoute2 stEntity stEnergy stPackets stRotation stNewPacket
neighborports : stRoute1 stRoute2 stRoute stEntity stEnergy stPackets stRotation stNewPacket

```

**Figure 5.** WSN energy model initialization definitions snippet.

```

% rule #1
rule : {~stRoute := $stRoute1; ~stRoute1 := $stRoute1; ~stRoute2:=
$stRoute2;#macro(setPortValue)}{$stRoute1 := -1; $stRoute2 := -1;#macro(setStatesValues) } 10
{$stEntity = 1}

% rule #2
rule : {~stRoute := $stRoute1; ~stRoute1 := $stRoute1; ~stRoute2:=
$stRoute2;#macro(setPortValue)}{$stRoute1 := 1; $stRoute2 := 1;#macro(setStatesValues) } 10
{$stEntity = 2}

```

**Figure 6.** First group rules in the WSN model specification.

### 3.1. WSN routing-related energy consumption model and simulation

The model in this section studies energy consumption with respect to nodes data forwarding and routing using a WSN architecture like the one discussed earlier in Figure 1. This section first presents a Cell-DEVS model specification (in section 3.1.1). It then simulates and visualizes the model with different scenarios (in section 3.1.2).

**3.1.1. Model specifications.** Each cell represents a geographical area in the space, which can contain up to one behavioral unit: a regular node, sink node, empty space, or an obstacle. Regular nodes are sensors that can collect and forward data toward sink nodes using the shortest path and data transmission amount to sense overloaded sensors. These factors are important to consider in order to preserve energy, hence the less work you do, the less energy to lose. Sink nodes are the edge nodes that receive all data to be forwarded toward external world. Empty space is a space that represents open area. Obstacles represent the blocking areas; hence, traffic needs to be routed around them.

During the model initialization phase (Figure 5), the rules initialize the routing tables for each node. Nodes may also use their neighboring nodes routing information in building its own tables. For each route in the table, a pointer to a neighboring node is set alongside alternative routes. After initialization is completed, data packets are generated at random time to simulate gathering data from the environment. Accordingly, packets are forwarded to

neighboring nodes according to the routing tables information (i.e. this is modeled by making receiving nodes check and collect their packets from sending nodes). The total number of processed and forwarded packets is used to determine the energy consumption by a node in proportionate with other nodes. This also indicates overloaded nodes (sensors) that might affect routing through them to slow their energy drain.

The initialization phase formal Cell-DEVS specification is shown in Figure 5. As shown in the specification, the model is defined as  $20 \times 20$  single plane cell space, defined by “dim: (20,20),” hence, creating 400 cells in the space. Each cell’s neighborhood contains nine cells, defined by “neighbors” keyword. Note that upon simulation start, the tools automatically convert all of these cells into DEVS atomic models and interconnect them with their applicable neighbors. Each cell uses eight state variables (keyword “statevariables”). These state values may indicate many things like cell type (i.e. intermediate node, sink node, empty, and obstacle). They also define other information like new packets forwarding, routes, energy calculation based on data transmission, and other visualization-related values. These states variables will soon be discussed alongside model rules discussion.

After the initialization phase, each cell (i.e. created as DEVS atomic model) behaves according to the rules discussed in the rest of this section. For simplicity, we discuss those rules in groups.

The first group is shown in Figure 6, initializes the nodes to be of the type of obstacle node, intermediate node, or of a sink (receiving) node type where sink nodes

```

#BeginMacro (setStatesValues)
$stRotation := remainder ($stRotation+1,10);
$stNewPacket := ifu($stRotation = 1 and $stRoute != 0 ,1,0,0);
$stPackets := if ( $stEntity != 1, $stNewPacket
+ ifu( (0,1)~stRoute = 4 , (0,1)~stPackets ,0,0)
+ ifu( (-1,0)~stRoute = 5 , (-1,0)~stPackets ,0,0)
+ ifu( (0,-1)~stRoute = 2 , (0,-1)~stPackets ,0,0)
+ ifu( (1,0)~stRoute = 3 , (1,0)~stPackets ,0,0),-1);
$stEnergy := if ( $stEntity != 1 , $stEnergy + $stPackets , 10000 );
#EndMacro

#BeginMacro (setPortValue)
~stRotation := $stRotation;
~stNewPacket := $stNewPacket;
~stPackets := $stPackets;
~stEnergy := $stEnergy;
#EndMacro

```

**Figure 7.** Defined macros in WSN model specification.

```

% rule #1
rule : {~stRoute := $stRoute1; ~stRoute1 := $stRoute1;#macro(setPortValue)}{$stRoute1 :=
2;#macro(setStatesValues) } 10 {$stRoute1 = 0 and (0,1)~stRoute1 >0}
% rule #2
rule : {~stRoute := $stRoute1; ~stRoute1 := $stRoute1;#macro(setPortValue)}{$stRoute1 :=
3;#macro(setStatesValues) } 10 {$stRoute1 = 0 and (-1,0)~stRoute1 >0}
% rule #3
rule : {~stRoute := $stRoute1; ~stRoute1 := $stRoute1;#macro(setPortValue)}{$stRoute1 :=
4;#macro(setStatesValues) } 10 {$stRoute1 = 0 and (0,-1)~stRoute1 >0}
% rule #4
rule : {~stRoute := $stRoute1; ~stRoute1 := $stRoute1;#macro(setPortValue)}{$stRoute1 :=
5;#macro(setStatesValues) } 10 {$stRoute1 = 0 and (1,0)~stRoute1 >0}

```

**Figure 8.** Second group rules in the WSN model specification.

are edge nodes, hence forward traffic to the outside world. This is coded as follows: if current cell is an obstacle/blocking node (i.e. *stEntity* is 1), then all routes (i.e. variables *stRoute*, *stRoute1*, and *stRoute2*) are set to  $-1$  (i.e. no routing is allowed). Accordingly, this node can also be viewed as a sleeping node. However, if current cell is a sink (i.e. *stEntity* is 2), then all routes are set to 1 (i.e. route to the external world).

As can be seen in the post-condition part of the two rules shown in Figure 6, there are two routine macros: *setStatesValues* and *setPortValue* we discuss following in Figure 7. Macro *setStatesValues* calculates a cell major four state variables as follows: *stRotation* (in the range of 0 through 9) is used to manipulate the rules execution order; *stNewPacket* indicates that a cell can receive new data packets (it is set to 1—true—for nodes in forwarding mode; otherwise, it is set to 0—false); *stPackets* is used to collect packets from neighboring nodes through the designated routes of those neighbors. It counts the number of queued packets in a node, which depends on if the cell can receive packets and the neighboring nodes have also

packets to transmit to this node. Finally, state variable *stEnergy*, which represents energy consumption by a node, is calculated based on the transmitted packets by a node. This also indicates overloaded nodes, hence, lose energy at faster rate comparing to other nodes. In this kind of cases, traffic tries to use alternative routes to reach sink nodes. The macro *setPortValue* is used by cells to set ports default values (unless those values are overwritten by the actual rules).

Now, if all rules in the first group (in Figure 6) are evaluated to false, the simulation moves on to execute the next rules, shown in Figure 8.

At this point in the rules (Figure 8), nodes are of the regular type, hence not of an empty, obstacle, or sink type. Therefore, shortest path routes to sink nodes need to be established if routes have not already been set. In this case, if a neighbor's route is already established, the current node then points to that neighbor route as a first step to establish a short path.

To be specific, in Figure 8, the first rule assigns the main route to point to neighbor (0,1) (if it is already set)



```

% rule #1
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
2;#macro(setStatesValues) } 10 {$stRoute2 =0 and $stRoute1 !=0 and $stRoute1 != 2 and
(0,1)~stRoute >0 and (0,1)~stRoute1 !=4 and (0,1)~stRoute2 !=4 }

% rule #2
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
3;#macro(setStatesValues) } 10 {$stRoute2 =0 and $stRoute1 !=0 and $stRoute1 != 3 and (-
1,0)~stRoute >0 and (-1,0)~stRoute1 !=5 and (-1,0)~stRoute2 !=5 }

% rule #3
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
4;#macro(setStatesValues) } 10 {$stRoute2 =0 and $stRoute1 !=0 and $stRoute1 != 4 and (0,-
1)~stRoute >0 and (0,-1)~stRoute1 !=2 and (0,-1)~stRoute2 !=2 }

% rule #4
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
5;#macro(setStatesValues) } 10 {$stRoute2 =0 and $stRoute1 !=0 and $stRoute1 != 5 and
(1,0)~stRoute >0 and (1,0)~stRoute1 !=3 and (1,0)~stRoute2 !=3 }

% rule #5
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
0;#macro(setStatesValues) } 10 {$stRoute2 =0 and $stRoute1 !=0 and ($stRoute1 = 2 or
(0,1)~stRoute =4) and ($stRoute1 =4 or (0,-1)~stRoute =2) and ( $stRoute1 =3 or (-1,0)~stRoute
=5)and ($stRoute1 =5 or (1,0)~stRoute =3)}

```

**Figure 9.** Third group rules in the WSN model specification.

by setting *stRoute1* to 2. Otherwise, the second rule assigns the main route to point to neighbor  $(-1,0)$  (if it is already set) by setting *stRoute1* to 3. Otherwise, the third rule assigns the main route to point to neighbor  $(0,-1)$  (if it is already set) by setting *stRoute1* to 4. Otherwise, the fourth rule in Figure 8 assigns the main route to point to neighbor  $(1,0)$  (if it is already set) by setting *stRoute1* to 5.

Now, if all rules in the second group (in Figure 8) are evaluated to false, the simulation moves on to execute the next rules, shown in Figure 9.

The rules in Figure 9 establish current node alternative route path. This is set to the neighboring node that has a path that can reach a sink node, hence shortest path to a sink node. It further ensures that this neighboring node path does not point back to the current node, hence avoiding loops in the routing paths.

To be specific, the *first rule* in Figure 9 assigns the alternative route to point to neighbor  $(0,1)$  (if a path reaches a sink and does not point back to current node) by setting *stRoute2* to 2. Otherwise, the *second rule* assigns the alternative route to point to neighbor  $(-1,0)$  (if path reaches sink and does not point back to current node) by setting *stRoute2* to 3. Otherwise, the *third rule* assigns the alternative route to point to neighbor  $(0,-1)$  (if path reaches sink and does not point back to current node) by setting *stRoute2* to 4. Otherwise, the *fourth rule* in Figure 9 assigns the alternative route to point to neighbor  $(1,0)$  (if path reaches sink and does not point back to current node) by setting *stRoute2* to 5. Finally, the *fifth rule* is a sanity check rule to make sure that there are no routing loops. If a loop is found, it is then removed by updating route values. It is worth noting that Rule 5 is extremely rare case, but it is worth a check.

At this point, when the rules shown in Figure 10 are reached, routing information has already been set for each node so that they can reach sink nodes via best possible shortest path. Thus, the rest of the rules are mainly concerned with choosing the best route through a neighbor to forward packets. Once a route is decided, the packets are forwarded and the node internal states (e.g. energy consumption) are updated accordingly, as previously discussed in macro *setStatesValues* in Figure 7. Routes selection for forwarding packets is based on energy consumption; hence, the route with lowest energy consumption is selected. In other words, routes with overloaded nodes can be avoided, if possible, to slow their energy drain. This provides load balancing throughout the network since energy consumption is computed based on the amount of transmitted data. This is important because energy efficiency can prolong nodes expected lifetime.

To be specific, *Rules 1–4* (in Figure 10) select a forwarding path for a packet if a neighboring cell has a reachable path to a sink node and the current node has consumed energy less of that neighboring node. *Rule 1* performs this check to neighbor  $(0,1)$  while Rules 2, 3, and 4 performs this check to neighbors  $(0,-1)$ ,  $(-1,0)$ , and  $(1,0)$ , respectively. However, if *Rules 1–4* have evaluated to false, then Rules 5–11 (in Figure 10) try to find the best forwarding path by comparing energy consumption between neighboring nodes. Rules 5–6, Rules 6–7, Rules 7–8, Rules 8–9, Rules 9–10, and Rules 10–11 perform energy comparison between neighbors  $\{(0,1)$  and  $(-1,0)\}$ ,  $\{(0,1)$  and  $(0,-1)\}$ ,  $\{(0,1)$  and  $(1,0)\}$ ,  $\{(-1,0)$  and  $(0,-1)\}$ ,  $\{(-1,0)$  and  $(1,0)\}$ , and  $\{(0,-1)$  and  $(1,0)\}$ , respectively.

```

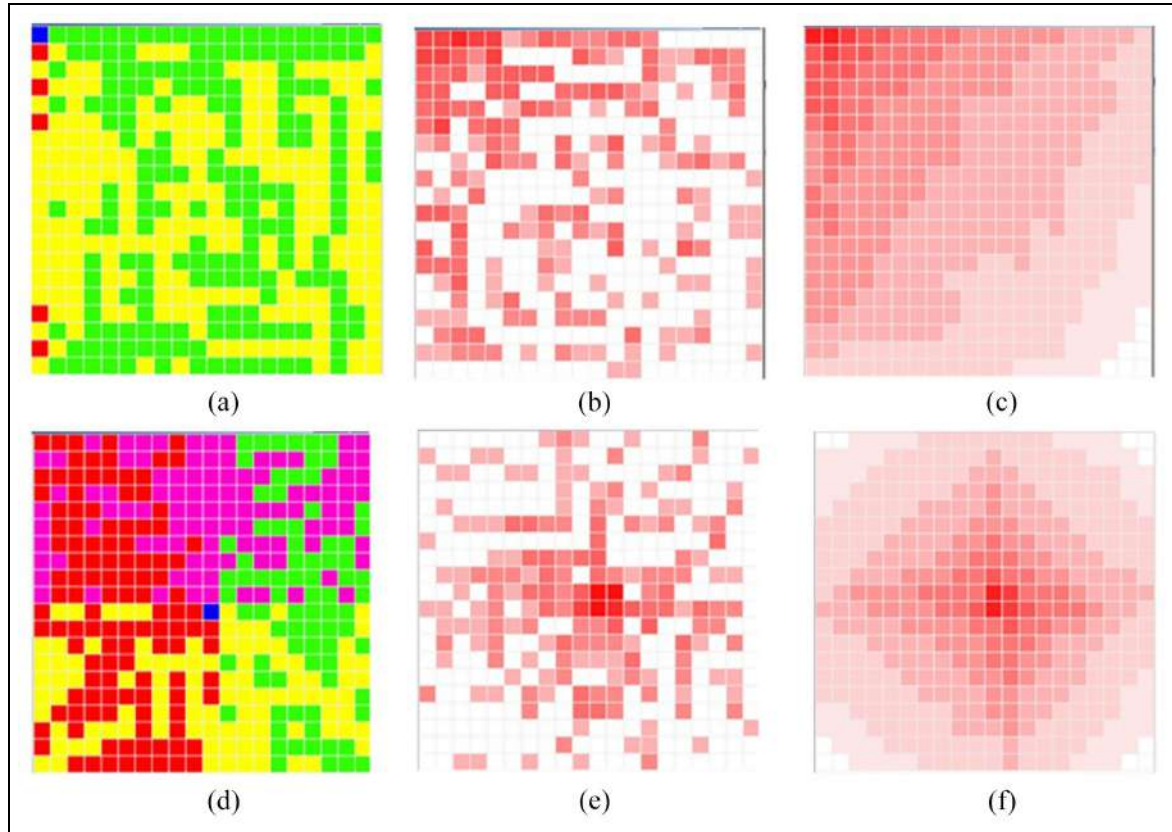
% *** Rule #1 ***
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
0;#macro(setStatesValues) } 10 { $stRoute2 =2 and ( (0,1)~stRoute1 =4 or (0,1)~stRoute2 =4) and
$stEnergy <(0,1)~stEnergy }
% *** Rule #2 ***
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
0;#macro(setStatesValues) } 10 { $stRoute2 =4 and ( (0,-1)~stRoute1 =2 or (0,-1)~stRoute2 =2) and
$stEnergy <(0,-1)~stEnergy }
% *** Rule #3 ***
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
0;#macro(setStatesValues) } 10 { $stRoute2 =3 and ( (-1,0)~stRoute1 =5 or (-1,0)~stRoute2 =5) and
$stEnergy <(-1,0)~stEnergy }
% *** Rule #4 ***
rule : {~stRoute := $stRoute1; ~stRoute2:= $stRoute2;#macro(setPortValue)}{$stRoute2 :=
0;#macro(setStatesValues) } 10 { $stRoute2 =5 and ( (1,0)~stRoute1 =3 or (1,0)~stRoute2 =3) and
$stEnergy <(1,0)~stEnergy }
% *** Rules 5-6 ***
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 2;#macro(setStatesValues) } 10 {((
$stRoute2 =2 and $stRoute1 =3) or ( $stRoute2 =3 and $stRoute1 =2)) and (0,1)~stEnergy < (-
1,0)~stEnergy }
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 3;#macro(setStatesValues) } 10 {((
$stRoute2 =2 and $stRoute1 = 3) or ( $stRoute2 =3 and $stRoute1 = 2)) }
% *** Rules 6-7 ***
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 2;#macro(setStatesValues) } 10 {((
$stRoute2 =2 and $stRoute1 = 4) or ( $stRoute2 =4 and $stRoute1 = 2)) and (0,1)~stEnergy < (0,-
1)~stEnergy }
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 4;#macro(setStatesValues) } 10 {((
$stRoute2 =2 and $stRoute1 = 4) or ( $stRoute2 =4 and $stRoute1 = 2)) }
% *** Rules 7-8 ***
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 2;#macro(setStatesValues) } 10 {((
$stRoute2 =2 and $stRoute1 = 5) or ( $stRoute2 =5 and $stRoute1 = 2)) and (0,1)~stEnergy <
(1,0)~stEnergy }
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 5;#macro(setStatesValues) } 10 {((
$stRoute2 =2 and $stRoute1 = 5) or ( $stRoute2 =5 and $stRoute1 = 2)) }
% *** Rules 8-9 ***
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 3;#macro(setStatesValues) } 10 {((
$stRoute2 =4 and $stRoute1 = 3) or ( $stRoute2 =3 and $stRoute1 = 4)) and (-1,0)~stEnergy < (0,-
1)~stEnergy }
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 4;#macro(setStatesValues) } 10 {((
$stRoute2 =4 and $stRoute1 = 3) or ( $stRoute2 =3 and $stRoute1 = 4)) }
% *** Rules 9-10 ***
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 3;#macro(setStatesValues) } 10 {((
$stRoute2 =5 and $stRoute1 = 3) or ( $stRoute2 =3 and $stRoute1 = 5)) and (-1,0)~stEnergy <
(1,0)~stEnergy }
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 5;#macro(setStatesValues) } 10 {((
$stRoute2 =5 and $stRoute1 = 3) or ( $stRoute2 =3 and $stRoute1 = 5)) }
% *** Rules 10-11 ***
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 4;#macro(setStatesValues) } 10 {((
$stRoute2 =4 and $stRoute1 = 5) or ( $stRoute2 =5 and $stRoute1 = 4)) and (0,-1)~stEnergy <
(1,0)~stEnergy }
rule : {~stRoute := $stRoute;#macro(setPortValue)}{$stRoute := 5;#macro(setStatesValues) } 10 {((
$stRoute2 =4 and $stRoute1 = 5) or ( $stRoute2 =5 and $stRoute1 = 4)) }
% *** Rule 12 ***
rule : {~stRotation := $stRotation;#macro(setPortValue)}{ #macro(setStatesValues) } 10 { t}

```

Figure 10. Fifth group rules in the WSN model specification.

Finally, Rule 12 is evaluated if all previous rules were evaluated to be false. This does not change the internal state variables of the current node (cell), however, the *stRotation* variable change, which affect the order of rules execution on the next simulation cycle.

**3.1.2. Model results.** The WSN energy model was simulated using CD++<sup>10</sup> with the focus on visualization of the system behavior under various scenarios to study energy consumption based on data transmission and shortest path routes to sink nodes. These routes need to avoid overloaded sensors to slow their energy drain.



**Figure 11.** Simulation snapshots for WSN with single sink node and without obstacles: (a) forwarding directions (sink on corner), (b) queued packets (sink on corner), (c) energy consumption (sink on corner), (d) forwarding directions (sink in middle), (e) queued packets (sink in middle), and (f) energy consumption (sink in middle).

Note that we will be discussing next simulation results from the following three dimensions (while visualizing the cell type like obstacles, intermediate node, and sink nodes):

1. The queued packets in nodes, hence, pending to be forwarded. This is the *stPackets* state variable in a node, as previously discussed in Figure 7. The more the queued packets in a node, the darker the assigned red color is.
2. The energy consumption by nodes. This is the *stEnergy* state variable in a node, as previously discussed in Figure 7. A darker red color is assigned when more energy is consumed by a node. Energy consumption is due to the previously discussed reasons.
3. The forwarding directions for packets in nodes (with the attempt to find shortest path while avoiding overloaded nodes). The current forwarding direction is indicated by state variable *stRoute* (see Figures 6–10), which can only be set to one of the

established routes *stRoute1* or *stRoute2* in the node routing table, as previously discussed. The cell is set to blue when *stRoute* is 1, which means a sink node. The cell is set to red when *stRoute* is 2, which means forwarding into the right direction. The cell is set to yellow when *stRoute* is 3, which means forwarding into the upward direction. The cell is set to green when *stRoute* is 4, which means forwarding into the left direction. The cell is set to pink when *stRoute* is 5, which means forwarding into the downward direction. The cell is set to black when *stRoute* is  $-1$ , which means blocking node, hence, obstacle in the network.

The rest of this section presents various simulation scenarios, which are simulated and presented to users by visualization. In these results, we selected a few snapshots of those scenarios visualization to be presented here.

**3.1.2.1. First case scenario.** Figure 11 shows the snapshots for the *first case scenario*. In this case, the WSN has

only *one sink node* to route traffic to the outside world. Furthermore, all cells in this case are acting as regular member nodes, which means that they can forward and route packets toward the sink node; hence, *no obstacles exist* in the network.

The top three snapshots in Figure 11 place the only sink node in the network on the top-left corner of the cell space, hence, the blue cell in Figure 11(a). Figure 11(a) shows the forwarding directions state for all nodes at time 80. As can be seen that cells are trying to reach the sink node by forwarding packets either toward their left (green) or toward their upward (yellow) directions. Figure 11(b) shows the number of pending packets for all nodes at time 80. The figure shows that most packets are queued in nodes close to the sink node (i.e. as shown by the darker red colored cells). Figure 11(c) shows the energy consumption map for all nodes at time 80. This figure clearly shows that the closer a node gets to the sink, the more energy it consumes (i.e. the darker the red color get). This makes sense since nodes closer to the sink node would drain more energy since they transmit and route more data than the far away nodes.

The above argument is further revealed in the bottom three snapshots in Figure 11. In this case, the single sink node is moved into the middle position; hence, it is shown as the blue cell in Figure 11(d). As shown the forwarding directions map (in Figure 11(d)), all of the nodes are forwarding toward the sink node in the middle {left (green), right (red), upward (yellow), and downward (pink)}. This also shown with the queued packets in nodes in Figure 11(e): nodes around the sink node area are queuing high volume of packets (i.e. darker red color). However, nodes that far away from the sink node can still queue packets at certain times. This means that those nodes are happened to be placed in an area with more data to collect at that time. Figure 11(f) shows that the closer a node gets to the middle (i.e. to the sink node), the more energy it consumes (i.e. shown as cells with darker red color). This is because nodes closer to the sink node route and transmit high volume of data, which consume more energy.

**3.1.2.2. Second case scenario.** Figure 12 shows the snapshots for *the second case scenario*. In this case, the WSN uses *multiple sink nodes* to route traffic to the outside world. Furthermore, all cells in this case are acting as regular nodes, which means that they can forward and route packets toward the sink nodes, hence *no obstacles* present in the network.

The top three snapshots in Figure 12 employ three sink nodes in the network where two of them placed on the top corners and the third one is placed in the middle-bottom row, as shown by the blue cells in Figure 12(a). Figure 12(a) shows the forwarding directions state for all nodes at time 80. As can be seen in Figure 12(a) that nodes are forwarding packets toward their closest sink node.

Accordingly, nodes close to left-top corner sink node forward packets into the left (green) and upward (yellow) directions. Similarly, nodes close to the top-right corner sink node forward packets into the right (red) and upward (yellow) directions. However, nodes in the bottom half forward packets to the middle-bottom sink node from different directions: downward (pink), right (red), and left (green) directions. Figure 12(b) shows the number of pending packets for all nodes at time step 80. The figure shows that most packets are queued in nodes close to the three sink nodes (i.e. as shown by darker red colored cells). Figure 12(c) shows the energy consumption map for all nodes at time 80. This figure clearly shows that the closer a node gets to a sink node, the more energy it consumes (i.e. the darker the red color get). This makes sense since nodes closer to the sink node would transmit and route more data than the far away nodes.

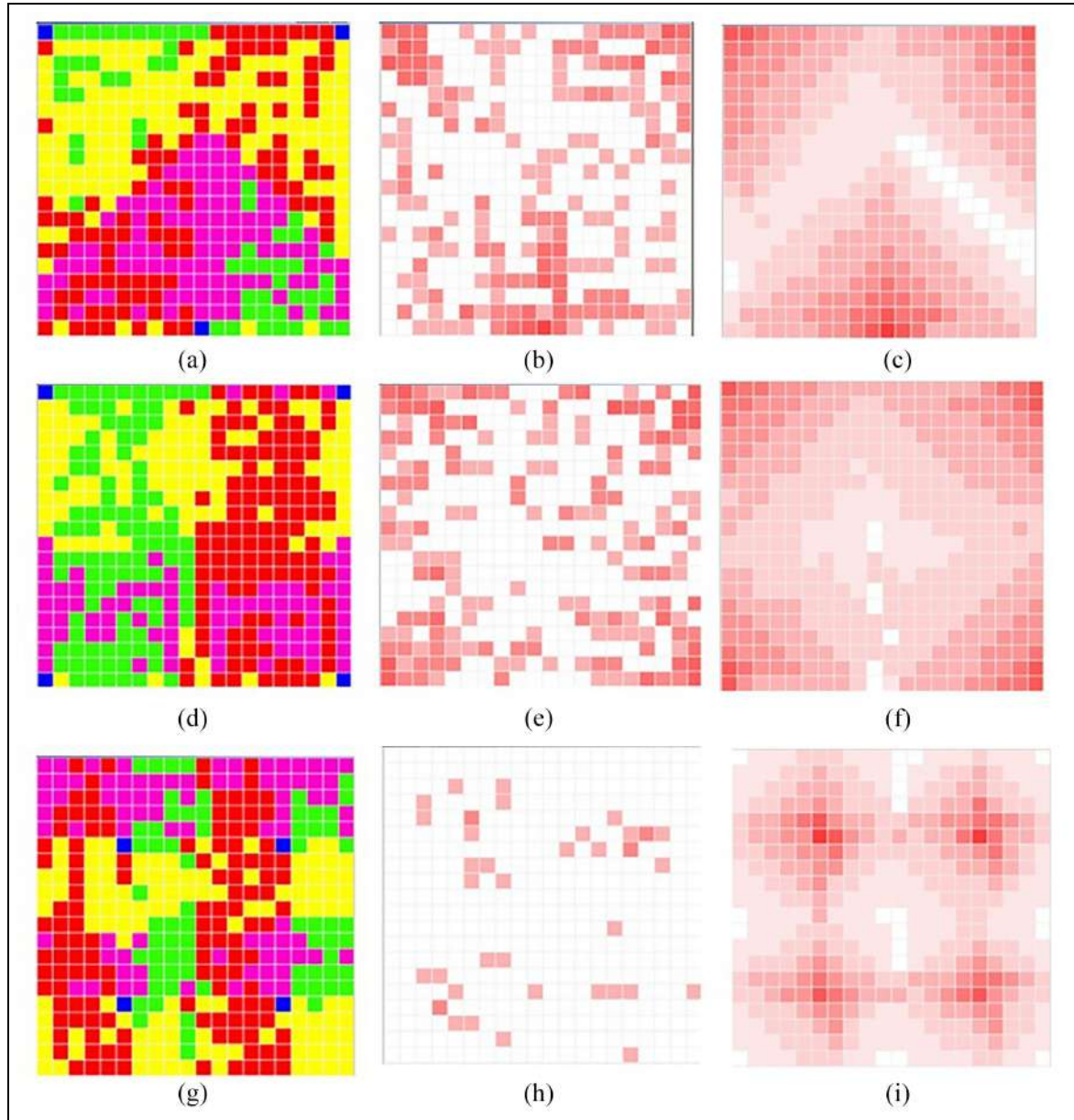
The above argument similarly applies to the three middle snapshots shown in Figure 12, which use four sink nodes where each is placed on a different corner. Figure 12(d) shows that nodes in each quarter of the cell space are forwarding packets to the sink node (i.e. shown in blue) that is placed in the corner closer to them. Figure 12(e) shows that most packets are currently queued in nodes closer to the corners (sink nodes). Furthermore, energy consumption is mostly occurring closer to the corners (Figure 12(f)), in a similar way to previous cases. However, this case has shown better energy efficiency for nodes located away from the corners.

However, the three bottom snapshots in Figure 12 show more interesting case. In this case, the four sink nodes are strategically positioned as square in the middle of the cell space, as shown by the blue cells in Figure 12(g). Figure 12(g) shows that most nodes can now reach sink nodes quicker than previous cases. This is clearly shown in Figure 12(h) where only low number of packets that get queued even with nodes around the sink nodes. This load balancing has improved energy efficiency in nodes as shown in Figure 12(i); hence, energy is mostly consumed by a few nodes around the sink nodes.

**3.1.2.3. Third case scenario.** The snapshots (at time step 80) in Figure 13 show *the third case scenario*. This case *introduces obstacles* in the network. Obstacles are modeled as cells (i.e. shown in black) without the capability of receiving or forwarding packets, hence packets then need to be routed around those obstacles to reach the sink nodes.

The top snapshots in Figure 13 show the case when one sink node is severely surrounded by obstacles. Figure 13(a) shows how nodes route to the sink node (i.e. blue cell) through the narrow top opening area. Figure 13(a) shows that bottom nodes mostly forward upward (yellow) while nodes near obstacles forward right (red) or left (green) to go around the obstacles. However, top nodes forward right

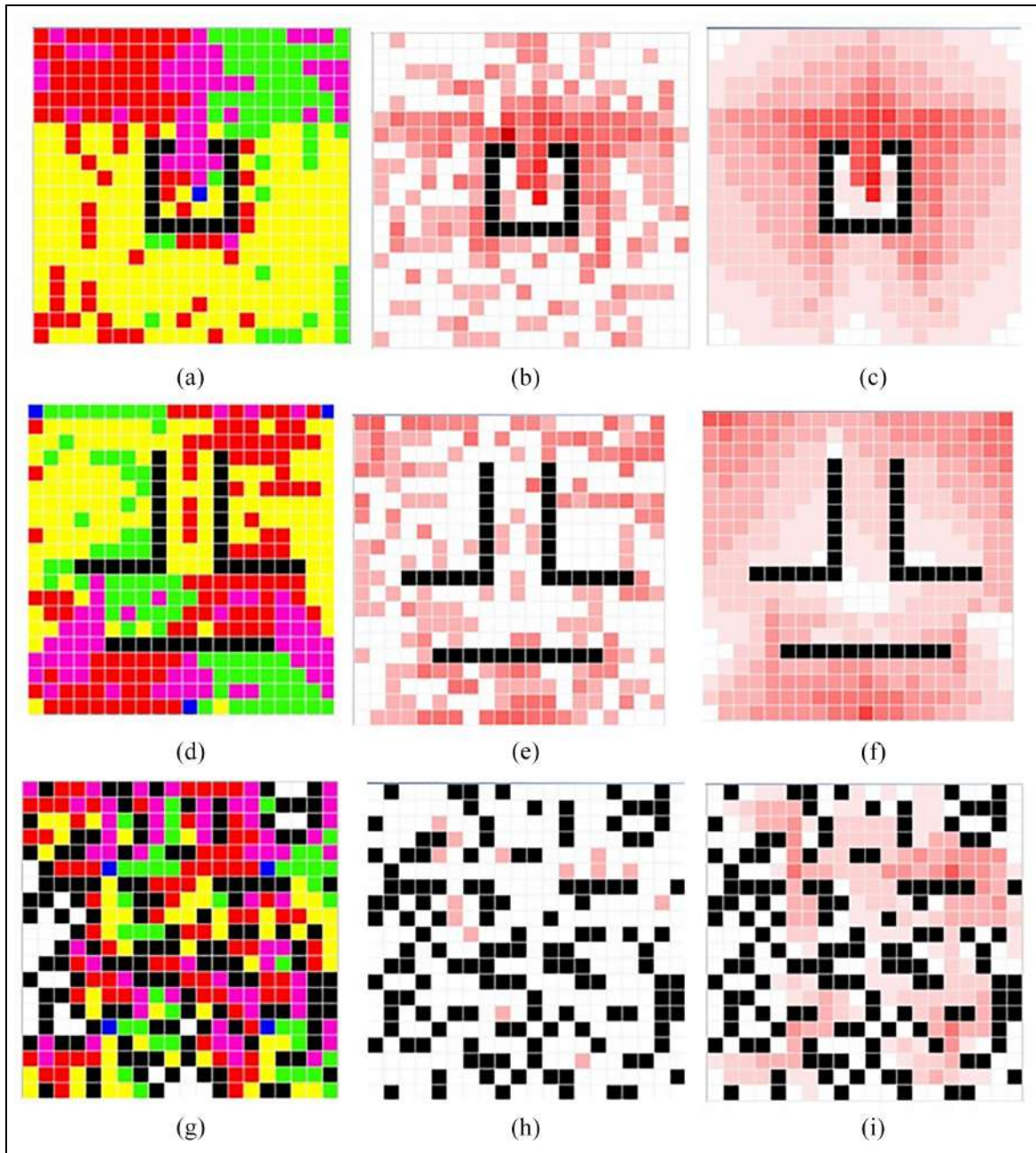




**Figure 12.** Simulation snapshots for WSN with multiple sink nodes and without obstacles: (a) forwarding directions (three sink nodes), (b) queued packets (three sink nodes), (c) energy consumption (three sink nodes), (d) forwarding directions (sinks on corners), (e) queued packets (sinks on corners), (f) energy consumption (sinks on corners), (g) forwarding directions (sinks in middle), (h) queued packets (sinks in middle), and (i) energy consumption (sinks in middle).

(red) or left (green) to find the narrow opening. However, nodes forward downward (pink) when the narrow opening area is found. Furthermore, most packets get queued near the narrow opening area, as shown by the darker red cells in Figure 13(b). Those nodes near the narrow opening area also consumed most energy, as shown by the darker red cells in Figure 13(c).

The same above argument also applies to the middle snapshots shown in Figure 13. This case shows obstacles built in a form of walls while using three sink nodes (i.e. blue cells in Figure 13(d)). Figure 13(d) shows how nodes route through the shortest path (around the obstacles) to reach its closest sink node (e.g. nodes forward to left (i.e. green) and upward (i.e. yellow) to reach left-top sink



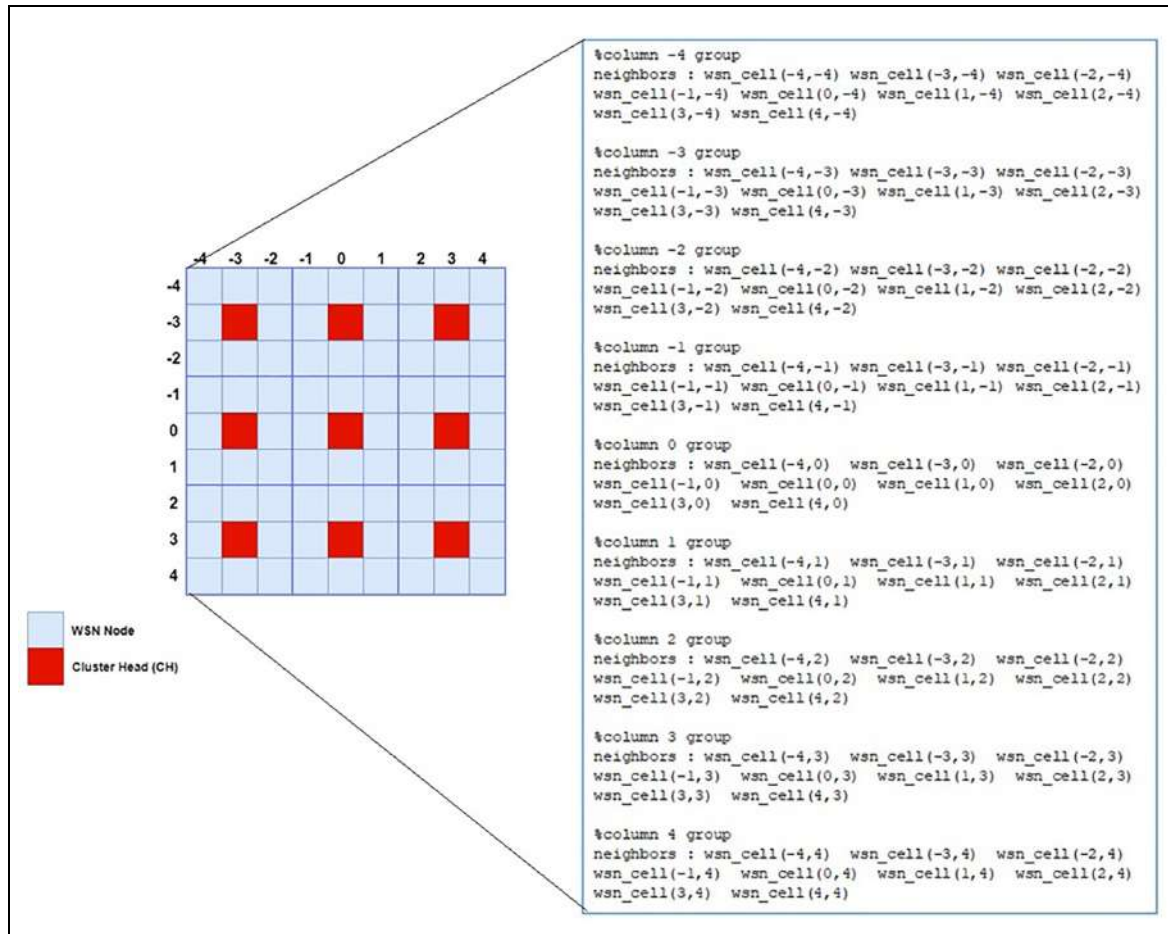
**Figure 13.** Simulation snapshots for WSN with obstacles: (a) forwarding directions (bad blocking), (b) queued packets (bad blocking), (c) energy consumption (bad blocking), (d) forwarding directions (blocking walls), (e) queued packets (blocking walls), (f) energy consumption (blocking walls), (g) forwarding directions (random blocking), (h) queued packets (random blocking), and (i) energy consumption (random blocking).

node). Figure 13(e) shows that most packets get queued around the sink nodes; hence, cells have darker red colors. Those nodes that surround the sink nodes also consume most energy (Figure 13(f)), as they do most data transmissions.

The bottom three snapshots in Figure 13 show more realistic case scenario. In this case, obstacles (i.e. black

cells) are randomly placed in the network. Furthermore, four sink nodes are strategically placed in the network, hence shown as the blue cells in Figure 13(g). Figure 13(g) shows how the nodes forward packets around obstacles to reach their closest sink nodes, hence forwarding is performed in all directions to go around those obstacles: green (left), red (right), yellow (upward), and pink (downward).





**Figure 14.** Topology control model neighborhood definition.

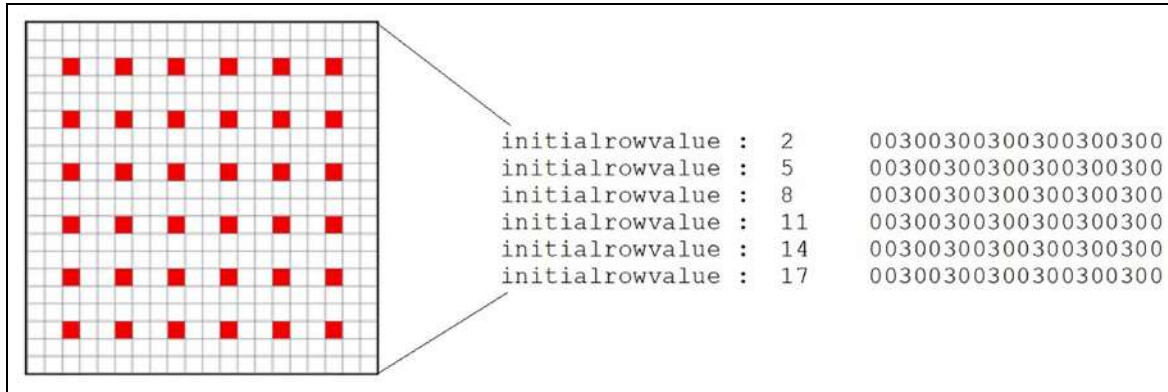
This led to better load balancing as shown by the low number of queued packets in Figure 13(h). This also shown by low energy consumption even by the nodes located around the sinks. This case clearly shows that careful placing of sink nodes can lead to better load balancing, which accordingly can lead to better energy efficiency, hence this would extend wireless nodes operating life in the WSN. The model shows that placing sink nodes strategically can improve data transmission distribution between nodes, hence improves energy efficiency.

### 3.2. WSN topology control energy M&S

The model in this section studies energy consumption at higher level, hence topology control method (using the WSN architecture previously discussed in Figure 2). In this model, nodes are organized in groups (called clusters). Each group selects one node, called CH, to communicate with other neighboring groups CH nodes. The model then controls energy consumption at the level of groups rather than at individual nodes themselves.

**3.2.1. Model specifications.** The topology control model is written in Cell-DEVS specification language and simulated by the CD++ tool. The model is defined as  $20 \times 20$  single plane cell space; hence, the model is assembled out of 400 DEVS atomic models since each cell is a separate DEVS atomic model. Model size can easily be changed without extra burden of the modeler. The neighborhood in this model is defined in groups (clusters) where each group contains a CH node to collect data from its group and to communicate with neighboring CH nodes as explained next in Figure 14.

Figure 14 (right) shows the neighborhood definition in the Cell-DEVS specification language. The specification simply lists all cells from cell  $(-4, -4)$  to cell  $(4, 4)$  in the neighborhood (i.e. this neighborhood space is visualized in Figure 14 (left)). The 81 cells in the neighborhood are divided into nine groups where each group has one CH node, placed in the middle as shown in Figure 14 (left). For example, the top-left cluster (in Figure 14 (left)) is defined by the square with corners of  $(-4, -4)$ ,  $(-4, -2)$ ,  $(-2, -4)$ , and  $(-2, -2)$  cells. In this way, each CH node



**Figure 15.** Topology control model initialization example.

has eight neighboring regular nodes within its cluster. It further has eight neighboring clusters; hence, it has eight CH neighbors since CH nodes act as surrogates for their clusters. Thus, the CH node shown at location (0,0) in Figure 14 (left) has eight CH neighbors located at  $(-3,-3)$ ,  $(-3,0)$ ,  $(-3,3)$ ,  $(0,-3)$ ,  $(0,3)$ ,  $(3,-3)$ ,  $(3,0)$ , and  $(3,3)$ . It is worth to note that during simulation each cell in the entire space treat itself as cell (0,0) when dealing with its neighborhood. Thus, each CH node sees its neighborhood in the same shown manner in Figure 14 (left).

As previously mentioned, Cell-DEVS uses state variables to store information in each cell, which play major role in defining cells behavior during simulation. The topology control model defines eight state variables: the node type (e.g. CH node and regular node), node residual energy, and node current communication operation (e.g. sending and receiving). Each of those values is also assigned a color to be used during simulation visualization. State variable Values (0–2) define residual energy levels for regular nodes while Values (3–8) define different state values for CH nodes. Value 3 (red color) indicates an active CH node, which means a switched-on node that can communicate with neighboring CH nodes, but not transmitting messages yet. Value 4 (black color) indicates an idle CH node, hence switched off to save energy. Value 5 (green color) indicates a CH node is being in the process of message transmission, hence data collected from its local group. Value 6 (purple color) indicates a CH node that is currently forwarding a message (which was originally sent by another CH) toward its destination CH. Value 7 (yellow color) indicates a CH node is currently acknowledging a received message. Value 8 (gray color) indicates message destination CH; hence, message is then forwarded by this node to the outside world.

The model starts with cell space initialization to place CH nodes. The placement of CH nodes in the space can vary from a use case scenario to another. For example, Figure 15 places 36 active CH nodes in the space. The

Cell-DEVS specification (Figure 15 (left)) places each active CH node (i.e. Value 3) in the desired position of the desired rows. Of course, CH nodes initial values (e.g. idle and sending) can vary depending on the used case scenario. It is worth noting that each of those CH nodes defines its neighborhood as previously discussed in Figure 14 (left).

After initialization phase, the internal simulation behavior is triggered in each node according to a set of rules. With each simulation cycle, rules are executed in order until one of those rules condition is satisfied. The first set of those rules is shown in Figure 16. Figure 16 (right) shows the Cell-DEVS specification snippet while Figure 16 (left) shows the conceptual flow of those rules. The focus of those rules is to control the duty cycle at the topology level to achieve better energy conservation. The idea is to decide when to turn off a CH node (and hence its local cluster) and to decide when to turn it back on. This is done as follows (Figure 16): the first rule sets energy level to random value, if the current node is a regular one. Otherwise, this node is CH, hence  $CD++$  simulator moves on executing the next rules in the specification. Rules 2, 3, and 4 (in Figure 16) only deal with CH nodes that are in the IDLE or ACTIVE states. Thus, if current CH is in a different state from those two, the simulator moves onto executing the next rules (will soon be discussed in Figure 17). Rules 2 and 3 set current CH node to IDLE if all of the followings are true: (1) current CH is ACTIVE or IDLE, and (2) has at least two CH neighbors in communication mode (i.e. state value  $\geq 5$ ), and (3) the average residual energy in the neighboring clusters is higher than the residual energy in the current CH local cluster. Finally, Rule 4 turns on current CH node (1) if the current is in IDLE state and (2) the average residual energy in the neighboring clusters is less than the residual energy in the current CH local cluster. Otherwise, the simulator moves onto executing the next rules shown in Figure 17.

The next rules are the communication rules (Figure 17) to mainly manage CH message exchanging states. Note

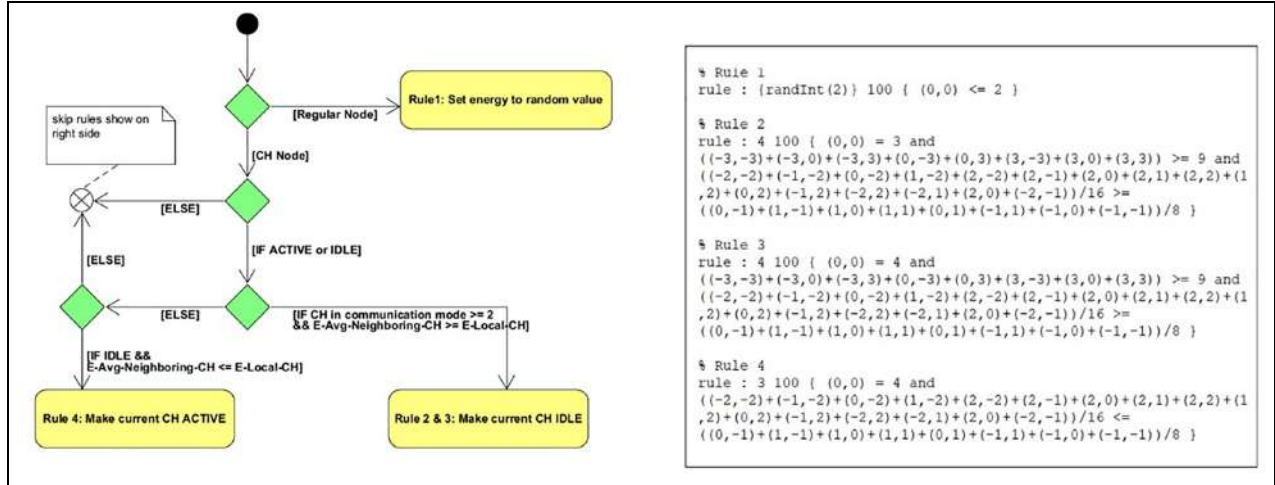


Figure 16. Topology energy control cycle rules.

```

%*** Rules 5-10 ***
rule : 6 100 { (0,0) = 3 and statecount(5) >= 1 }
rule : 6 100 { (0,0) = 3 and statecount(6) >= 1 }
rule : 7 100 { (0,0) = 6 and statecount(8) >= 1 }
rule : 3 100 { (0,0) = 5 and statecount(7) >= 1 }
rule : 7 100 { (0,0) = 6 and statecount(7) >= 1 }
rule : 3 100 { (0,0) = 8 and statecount(6) >= 1 }
%*** Rules 11-13 ***
rule : 5 100 { (0,0) = 5 }
rule : 6 100 { (0,0) = 6 }
rule : 8 100 { (0,0) = 8 }
%*** Rule 14 ***
rule : 3 100 { t }

```

Figure 17. CH messaging rules example.

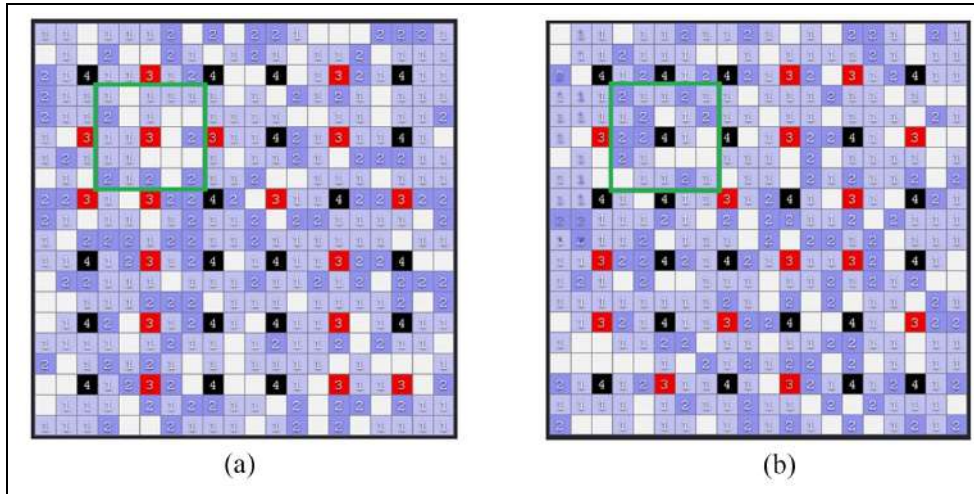
that these rules could be changed to study different use cases scenarios, as desired. Figure 17 shows an example of CH message exchanging rules. Rules 5–10 (in Figure 17) summarized as follows: CH node becomes in *Sending Message* state (i.e. Value 5) when it has a message to send. This message represents collected data from the sender CH local neighborhood. CH in Active state becomes in *Forwarding Message* (i.e. Value 6) if it gets a message from a neighbor in the *Sending Message* state or in the *Forwarding Message* state. CH in the *Receiving message* state (i.e. Value 8) becomes *Active* when it receives the message. This represents message being forwarded outside the WSN. CH becomes in the *Acknowledging Message* state (i.e. Value 7) if it gets an ACK from a neighbor in the *Receiving Message* state or in the *Acknowledging Message* state. CH in *Sending Message* state (i.e. Value 5) becomes *Active* (i.e. Value 3) when it gets an ACK from a neighbor CH. Rules 11–13 (in Figure 17) do not change a CH node state if there are no communicating neighbors and this CH in one of the following states: *Sending*

*Message*, *Forwarding Message*, or *Receiving message* state. Finally, if all above rules are evaluated to false, the default rule (i.e. Rule 14 in Figure 17) always puts the CH node into the *Active* state.

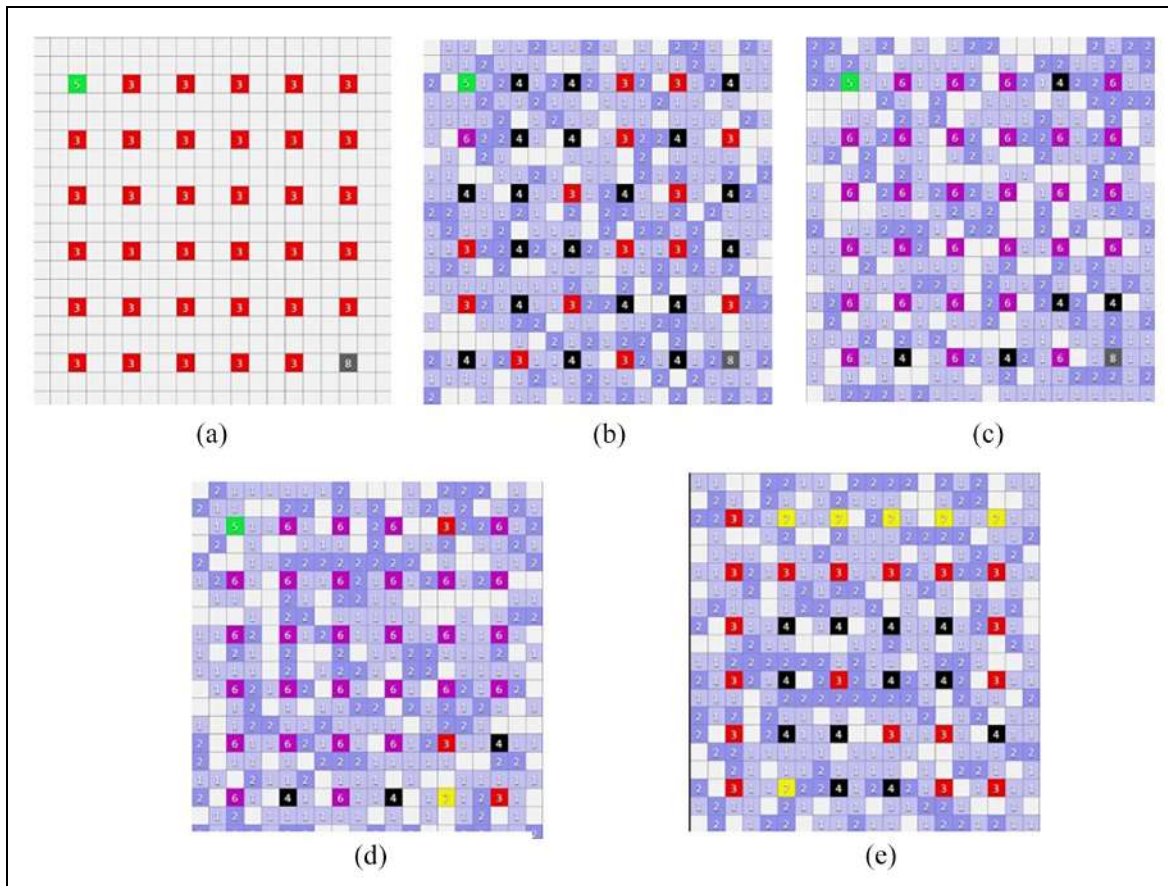
**3.2.2. Model results.** The topology control model was simulated using CD++ with the focus on observing CH nodes behavior from their neighborhood perspectives.

For example, Figure 18(a) shows a simulation results snapshot at time 00:00:05:00. As can be seen, the CH node in the middle of the green rectangle is active (red), hence it is currently turned on. The eight neighboring CH nodes are shown on the green rectangle borders. Some of those neighbors are active (red) while others are idle (black). Now when the rules (discussed in Figure 16) are executed for the CH node in the middle of the green rectangle, it first finds out that it has more than two active CH neighbors. It then calculates the average residual energy for the CH neighbors to be 1.0625 while the residual energy for the local neighborhood (i.e. nodes inside the green rectangle) is calculated to be 0.375. Based on this, the topology control method turns this CH node off, which is shown as black (idle) cell in the simulation next step (in Figure 18(b)). The simulation was executed with various scenarios by placing CH senders and receivers on different positions. However, since those scenarios have shown similar results, it should be sufficient to show only one scenario here. In this scenario, one CH is collecting data from its local neighborhood (i.e. the sensors in its cluster). This sender CH is the top-left green cell in Figure 19(a) while the receiver CH node is the right-bottom gray cell in Figure 19(a). The sender CH node data are continuously forwarded via middle CH nodes until data reach the receiving (destination) CH node. Figure 19(b) snapshot shows when the sender CH sends its first message via one of its





**Figure 18.** Simulation snapshots example: (a) simulation results (first step) and (b) simulation results (next step).



**Figure 19.** Simulation messages transmission scenario example: (a) initial states, (b) initial sending/forwarding, (c) during sending/forwarding, (d) complete message receipt, and (e) complete message acknowledgment.

CH neighbors. This CH neighbor (Figure 19(b)) is shown in purple color cell, hence in forwarding state. Figure 19(b) also shows CH local neighborhoods with different random residual energy levels. It further shows that many

CH nodes have been turned off (i.e. black cells) to save energy as part of the topology control method. Figure 19(c) snapshot shows that more CH nodes are involved in forwarding messages (i.e. purple cells) between the sender

and the receiver CH nodes. Figure 19(d) snapshot shows the step when the receiver CH has acknowledged the received message via one of its CH neighbors. As a result, the figure shows that the original receiver is now in the active state (i.e. red color) while the CH neighbor that is currently forwarding the ACK back to the sender CH is in the acknowledgment state (i.e. yellow color). Figure 19(e) snapshot shows that the sender CH has received the last ACK, hence changed state to active (i.e. red color). This figure shows that many CH nodes have been turned off (i.e. black cells) to save energy while others still turned on (i.e. red cells) to be able to communicate in the future if needed. The figure also shows that some CH nodes still forwarding ACKs (i.e. yellow cells). This represents duplicate ACKs that will be ignored by the sender CH.

## 4. Conclusion

WSN applications have been widely used in recent years, in particular with the advent of Internet of Things (IoTs), which in many cases need to detect and collect information from environments. Regardless of the used M&S environment (e.g. NS-2, J-Sim, OPNET, and OMNet++), WSN sensors are usually defined as a behavioral model (or a simulation component usually written in a programming language) while the network is defined as a collection of interconnected nodes using a scripting language, a GUI, etc. We presented an alternative method for modeling WSN, based on the formal specification of the WSN using the Cell-DEVS formalism: the space is partitioned into cells where each cell can be considered a sensor, an obstacle, or anything of a behavior with defined rules. These models consider the spatial location of the nodes, making it useful for spatial analysis of the system under study. We further showed how those models could be simulated and visualized. The models covered energy-related issues at the sensor (node) level and when sensors organized in multiple clusters. We considered energy consumption to be related to amount of data being transmitted, which affect the routing protocols to use shortest path to reach sink nodes while trying to avoid overloaded nodes. The simulation results that placing major nodes like sink and CH nodes strategically can improve energy efficiency in WSN, hence prolong its lifetime.


## Acknowledgements


The authors want to acknowledge the participation of various students who developed some of the models presented here, including M. Shabani and V. Soto. The complete models with original source code can be found at [https://cell-devs.sce.carleton.ca/mediawiki/index.php/Model\\_Samples](https://cell-devs.sce.carleton.ca/mediawiki/index.php/Model_Samples).

## Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was partially funded by the NSERC, Natural Sciences and Engineering Research Council of Canada.

## ORCID iDs

Khaldoon Al-Zoubi  <https://orcid.org/0000-0001-6194-3449>

Gabriel A Wainer  <https://orcid.org/0000-0003-3366-9184>

## References

1. Ullo SL and Sinha GR. Advances in smart environment monitoring systems using IoT and sensors. *Sensors* 2020; 20: 3113.
2. Fu X, Yao H, Postolache O, et al. Message forwarding for WSN-assisted opportunistic network in disaster scenarios. *J Netw Comput Appl* 2019; 137: 11–24.
3. Fu X, Yao H and Yang Y. Cascading failures in wireless sensor networks with load redistribution of links and nodes. *Ad Hoc Netw* 2019; 93: 101900.
4. Yi S, Li C and Li Q. A survey of fog computing: concepts applications and issues. In: *Proceedings of the 2015 workshop mobile big data*, Hangzhou, China, 21 June 2015.
5. Al-Zoubi K and Wainer G. Fog and cloud collaboration to perform virtual simulation experiments. *Simul Model Pract Th* 2020; 101: 102032.
6. Network Simulator (NS-2), <https://www.isi.edu/nsnam/ns/> (accessed 5 January 2022).
7. Java Simulator (J-Sim), <https://www.kiv.zcu.cz/j-sim/> (accessed 23 April 2022).
8. OPNET network simulator, <https://opnetprojects.com/opnet-network-simulator/> (accessed 5 January 2022).
9. OMNet++ simulator, <https://omnetpp.org/> (accessed 5 January 2022).
10. Wainer GA. *Discrete-event modeling and simulation: a practitioner's approach*. Boca Raton, FL: CRC/Taylor & Francis, 2009.
11. Jun L and YinSong W. Gradient-based clustering routing algorithm for EH-WSN in transmission line monitoring. In: *Proceedings of the 39th Chinese control conference (CCC)*, Shenyang, China, 27–29 July 2020.
12. Bahbahani MS and Alsusa E. A cooperative clustering protocol with duty cycling for energy harvesting enabled wireless sensor networks. *IEEE T Wirel Commun* 2018; 17: 101–111.
13. Pegatoquet A, Le TN and Magno M. A wake-up radio-based MAC protocol for autonomous wireless sensor networks. *IEEE ACM T Network* 2019; 27: 56–70.
14. Zhang D, Wang Y, Lv Y, et al. A cluster head multi-hop topology control algorithm for WSN. In: *Proceedings of the 2019 IEEE 2nd international conference on computer and communication engineering technology (CCET)*, Beijing, China, 16–18 August 2019.
15. Gamal M, Sadek N, Rizk M, et al. Markov model of modified unslotted CSMA/CA for wireless sensor networks. In:

- Proceedings of the 31st international conference on micro-electronics (ICM)*, Cairo, 15–18 December 2019.
16. Shemim KSF and Witkowski U. Energy efficient clustering protocols for WSN: performance analysis of FL-EE-NC with LEACH, K Means-LEACH, LEACH-FL and FL-EE/D using NS-2. In: *Proceedings of the 32nd international conference on microelectronics (ICM)*, Aqaba, Jordan, 14–17 December 2020.
  17. Shemim KSF and Witkowski U. Energy efficient clustering protocols in WSNs: performance analysis and comparison of EEHP protocol with LEACH and EAMMH using MATLAB. In: *Proceedings of the 2020 advances in science and engineering technology international conferences (ASET)*, Dubai, United Arab Emirates, 4 February–9 April 2020.
  18. Abbas SH and Khanjar IM. Fuzzy logic approach for cluster-head election in wireless sensor network. *Int J Eng Res Adv Technol* 2019; 5: 14–25.
  19. Accha V and Gupta SH. Performance analysis of Wireless Sensor Network MAC protocols using NS-2. In: *Proceedings of the 2018 international conference on computing, power and communication technologies (GUCON)*, Greater Noida, India, 28–29 September 2018.
  20. Sobeih A, Hou JC, Kung L-C, et al. J-Sim: a simulation and emulation environment for wireless sensor networks. *IEEE Wirel Commun* 2006; 13: 104–119.
  21. Neves PACS, Veiga IDC and Rodrigues JJPC. G-JSIM—a GUI tool for Wireless Sensor Networks simulations under J-SIM. In: *Proceedings of the 2008 IEEE international symposium on consumer electronics*, Vilamoura, 14–16 April 2008.
  22. Nam SM and Kim HJ. WSN-SES/MB: system entity structure and model base framework for large-scale wireless sensor networks. *Sensors* 2021; 21: 430.
  23. Alsaif O, Saleh I and Ali D. Evaluating the performance of nodes mobility for Zigbee Wireless Sensor Network. In: *Proceedings of the 2019 international conference on computing and information science and technology and their applications (ICCISTA)*, Kirkuk, Iraq, 3–5 March 2019.
  24. Robinson Y, Krishnan RS, Narayanan KL, et al. Hybrid data forwarding technique for enhanced lifetime in Wireless Sensor Networks. In: *Proceedings of the 5th international conference on trends in electronics and informatics (ICOEI)*, Tirunelveli, India, 3–5 June 2021.
  25. Kodali RK and Malothu VK. MIXIM framework simulation of WSN with QoS. In: *Proceedings of the 2016 IEEE international conference on advanced communication control and computing technologies (ICACCCT)*, Ramanathapuram, India, 25–27 May 2016.
  26. Kashyap VK, Astya R, Nand P, et al. Comparative study of AODV and DSR routing protocols in wireless sensor network using NS-2 simulator. In: *Proceedings of the 2017 international conference on computing, communication and automation (ICCCA)*, Greater Noida, India, 5–6 May 2017.
  27. ZigBee cluster library user guide, <https://www.nxp.com/docs/en/user-guide/JN-UG-3077.pdf> (accessed 7 January 2022).
  28. Tavanpour M, Kazi BU and Wainer G. Discrete event systems specifications modelling and simulation of wireless networking applications. *J Simul* 2022; 16: 1–25.
  29. Sabor N, Sasakia S, Abo-Zahhad M, et al. A graphical-based educational simulation tool for Wireless Sensor Networks. *Simul Model Pract Th* 2016; 69: 55–79.
  30. Valentine DT and Hahn B. *Essential MATLAB for engineers and scientists*. 7th ed. Amsterdam: Elsevier, 2019.
  31. Gupta S, Mittal M and Padha A. Predictive analytics of sensor data based on supervised machine learning algorithms. In: *Proceedings of the 2017 international conference on next generation computing and information systems (ICNGCIS)*, Jammu, India, 11–12 December 2017.
  32. Al-Zoubi K and Wainer G. Distributed simulation of DEVS and Cell-DEVS models using the RISE middleware. *Simul Model Pract Th* 2015; 55: 27–45.
  33. Al-Zoubi K and Wainer G. Mobile experimentation using modelling and simulation in the Fog/Cloud. *J Simul*. Epub ahead of print 20 August 2021. DOI: 10.1080/477778.2021.1964393.

### Author biographies

**K.A.-Z.** received both PhD (2011) in Electrical and Computer Engineering and MCS (2006) from Carleton University (Ottawa, ON, Canada). He received a BSc in Electrical and Computer Engineering (1995) from the University of Louisiana at Lafayette (Lafayette, LA, USA). Before joining the faculty of Computer Information Technology at the Jordan University of Jordan (JUST) as assistant professor in 2016, he worked for more than 20 years as a Senior Software Engineer, Developer, Researcher, and Inventor in leading hi-tech companies (e.g. Huawei, BlackBerry, and Nav Canada) in Canada and the United States. During his industry experience, he had full hands-on involvement in generating patents (for real-world products) and in architecting, designing, and developing wide range of software solutions that have been used by hundreds of millions of people around the world. This industry experience and products have covered different fields mainly in networking, 5G-based Data centers, Software Defined Networks (SDNs), large-scale Cloud Computing, embedded software, Real-Time Systems, Graphical User Interface (GUI), Device Drivers, Client/Server communication, QNX/Linux, Explosives/ Narcotics detections, and air-traffic communication systems. In addition, his academic research has been involved in Fog and Cloud Computing, Cloud-based Modeling and Simulation as services, Mobile Computing, Wireless and Networking protocols, Web-services, and Parallel and Distributed Simulation, and Cyber Security.

**G.A.W., FSCS**, received the MSc (1993) at the University of Buenos Aires, Argentina, and the PhD (1998, with highest honors) at UBA/Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada), where he is now Full Professor. He has held visiting positions at the



University of Arizona; LSIS (CNRS), Université Paul Cézanne, University of Nice, INRIA Sophia-Antipolis, Université de Bordeaux (France); UCM, UPM, UPC (Spain), University of Buenos Aires, National University of Rosario (Argentina), and others. He is one of the founders of SIMUTools, ANNSIM (SCS/IEEE/ACM), the Symposium on Theory of Modeling and Simulation (SCS/ACM/IEEE), and Symposium on Simulation in Architecture and Urban Design—SimAUD (SCS/ACM/IEEE). Professor Wainer is Editor in Chief of SIMULATION, member of the Editorial Board of Journal of Simulation, IEEE Computing in Science and Engineering, Wireless Networks and Journal of Defense Modeling and Simulation (SCS). He is the head of the

Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He has been the recipient of various awards, including the IBM Eclipse Innovation, SCS Leadership, and various Best Papers. He has been awarded Carleton University's Research Achievement Award (2005 and 2014), the SCS Outstanding Professional Award (2011), Carleton University's Mentorship Award (2013), the SCS Distinguished Professional Award (2013), the SCS Distinguished Service Award (2015), Nepean's Canada 150th Anniversary Medal (2017), ACM Recognition of Service Award (2018), and IEEE Outstanding Engineering Award (Ottawa Section—2019). He is an ACM Distinguished Speaker and a Fellow of SCS.