Available online at www.sciencedirect.com
ScienceDirect
Simulation Modelling Practice and Theory 15 (2007) 285–314
SIMULATION MODELLING PRACTICE AND THEORY
www.elsevier.com/locate/simpat

DEVS modeling of mobile wireless ad hoc networks

Umar Farooq, Gabriel Wainer *, Bengu Balya

Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Dr., Ottawa, ON, Canada K1S 5B6

Received 8 December 2005; received in revised form 3 November 2006; accepted 7 November 2006
Available online 20 December 2006

Abstract

Ad hoc networks are self-organizing wireless systems conformed by cooperating neighboring nodes that conform networks with variable topology. Analyzing these networks is a complex task due to their dynamic and irregular nature. Cellular Automata (CA), a very popular technique to study self-organizing systems, can be used to model and simulate ad hoc networks, as the modeling technique resembles the system being modeled. Cell-DEVS was proposed as an extension to CA in which each cell in the system is considered as a DEVS model. The approach permits defining models with asynchronous behavior, and to execute them with high efficiency. We show how these techniques can be used to model mobile wireless ad hoc networks, making easy model definition, analysis and visualization of the results. The use of Cell-DEVS permitted us to easily develop new experiments, which allowed us to extend routing techniques for inter-networking and multicast routing, while permitting seamless integration with traditional networking models.
© 2006 Published by Elsevier B.V.

Keywords: Cellular models; Wireless ad hoc networks; DEVS; Cell-DEVS

1. Introduction

Mobile ad hoc networks are conformed by transitory association of mobile nodes. They do not require any fixed support infrastructure and are typically based on short-range wireless technology. Mobile ad hoc networks are generally characterized by their highly dynamic nature, varying topology and intermittent connectivity. One of the important and challenging aspects in the study of ad hoc networks is the routing algorithm used for data transmission between nodes. This is a challenging task because of their lack of infrastructure and highly dynamic nature. Any routing algorithm has to take into account the varying topology of the network and unreliable and generally bandwidth-limited wireless links. Modeling and simulation has been widely used to analyze these complex characteristics of mobile ad hoc networks, and to study the execution results of new routing algorithms [1].

Different modeling and simulation tools have been applied to analyze these networks [2–4], but in recent years, different authors [5,6] decided to represent this problem using Cellular Automata [7]. The Cellular Auto-

* Corresponding author. Tel.: 1 613 520 2600.
E-mail addresses: ufarooq@sce.carleton.ca (U. Farooq), gwainer@sce.carleton.ca (G. Wainer), bengu@sce.carleton.ca (B. Balya).

1569-190X/$ - see front matter © 2006 Published by Elsevier B.V.
doi:10.1016/j.simpat.2006.11.011

mata technique permits providing good means for describing complex systems like these ones while being able to analyze their spatial characteristics. Cellular Automata are discrete-time discrete models described as cells organized as n-dimensional infinite lattices. Each cell in the automaton has a discrete value that is changed by a local computation function, whose results are computed locally in each cell using the present value for the cell and a finite set of neighbors. The use of such a discrete time base may constrain the precision of the model. In addition, the execution of cellular automata usually requires a large amount of computation time, primarily due to its synchronous nature. Cell-DEVS [8] solves these problems using the DEVS (Discrete Events systems Specification) formalism [9] to define a cell space where each cell is defined as a DEVS model. Cell-DEVS permits building discrete-event cell spaces and improves their definition by making their timing specification more expressive.

DEVS allows the modeler to formally specify discrete-event systems using modular descriptions. This strategy allows the reuse of tested models, improving the safety of the simulations and allowing reducing the development times. As it uses a continuous time base, precision of the models can be improved while CPU time requirements can be reduced. Higher timing precision can be obtained without using small discrete time segments (that would increase the number of simulation cycles). The formalism is based on sound theoretical grounds, allowing for an abstract design of models that are independent from the implementation platform. The use of DEVS and Cell-DEVS provides:

- Facilities to carry out formal tests.
- Seamless model sharing between different DEVS-based toolkits [10].
- High-performance execution of the same models in a parallel simulation environment [11].
- Remote execution using client–server services, allowing remote interaction between users [12].
- The ability to execute the models on a distributed platform based on HLA [13,14], CORBA [15] or other technologies.
- The possibility to define models using different techniques interacting within the same environment [16]. This could allow including non-network entities that affect network operation, providing results that are more realistic.
- The potential to automatically deploy models that have been tested on the simulation environment into the actual networking hardware, converting them into the real applications [17,18].

The CD++ toolkit [10] implements DEVS and Cell-DEVS theories. The toolkit has been built as a set of independent software pieces, each of them independent of the operating environment chosen. Our goal was to build models of ad hoc networks using the advantages of DEVS and Cell-DEVS for the following reasons: efficiency (by describing a high level specification of the problem to be modeled, we have reduced the effort needed in developing the application) and high performance (the models execute using a discrete-event approach, which, as shown in [18–20], provide higher precision and speedups than the discrete time approaches). In [18], the authors showed that the discrete-event nature of DEVS models combined with parallel simulation techniques can produce speedups of up to 1000 times. In [19,20], we showed that Cell-DEVS also provides these advantages. DEVS also provides a formal framework that can be used to validate and verify the models, opening the door to re-using the models as well as integrating them with other models based on different formalisms (for instance, using Petri Nets or Finite State Machines to specify the behavior of traffic lights or railway controllers). System specification can be done in a simple fashion, without spending time in coding or testing every proposed solution to existing problems, and new rules can be easily incorporated, as we will show with different examples here.

We aim to demonstrate the applicability, advantages and limitations of DEVS and Cell-DEVS techniques in modeling and simulating wireless ad hoc networks. The results presented in this paper serve as a proof-of-concept of the feasibility of our proposal. As the tools we use and the models we present are open-source, this research can be used as a basis for future research efforts in the field. Therefore, we decided to include a detailed version of many of the models, allowing the reader to fully understand how models are created. We begin by using a variant of classical Lee's Algorithm [21] to find out the shortest path between two communicating nodes in an ad hoc network plane. We show how such algorithms can be implemented in Cell-DEVS easily and efficiently. We then show how to model one of the most widely accepted routing protocol

for wireless ad hoc networks, Ad Hoc On-Demand Distance Vector Protocol (AODV) [22] using cell-DEVS. We then extend AODV for inter-network routing by making use of a three-dimensional Cell-DEVS model. We also extend unicast AODV algorithm to a multicasting algorithm while achieving the optimality in multicast tree construction (i.e., the multicast trees are constructed in such a way that ensures least duplication of data). We then present a model for routing using AODV among multiple pairs of senders and receivers. Hochberger [23] has shown that if there are multiple pairs of senders and receivers on a cellular plane, it may generate deadlocks and may prevent the generation of routing path between pairs of nodes that can communicate. We found a simple solution to the problem by exploiting the inherent parallelism in Cell-DEVS. We also demonstrate the use of Cell-DEVS in modeling mobile nodes in an ad hoc network for the scenarios where each node determines and regularly updates its shortest hop count from the wireless gateway. This information can help forwarding the data to the neighbor that has the smallest number of hops to the gateway. Finally, we model network coverage which is an important parameter for network engineers and can help in determining the suitable location for the installation of gateways. As nodes can be either static or mobile, we have implemented collision avoidance techniques.

2. Background

Wireless ad hoc networks are characterized by dynamic topology changes, severe power constraints and unpredictable wireless environments. There is no infrastructure and each node acts as a router to forward traffic. This means routers are mobile themselves, and face all other challenges such as dynamic topology changes, unreliable links and power constraints. Nodes are connected with wireless links that are very prone to environment factors such as fading, shadowing and noise. Therefore, link failures and congestion are the normal characteristics of the network, and should not be considered as exceptions. This makes routing in ad hoc networks quite a challenging task.

Several routing algorithms have been proposed, such as Destination-Sequenced Distance Vector Protocol (DSDV) [24], Dynamic Source Routing (DSR) [25], Ad Hoc On-Demand Distance Vector Protocol (AODV) [21] and Temporally Ordered Routing Algorithm (TORA) [26]. DSDV, just like most of the traditional routing algorithms, is a table driven routing algorithm. On the other hand, DSR, TORA and AODV are source initiated on-demand algorithms. Each algorithm has its advantages and disadvantages. In our study, we have used one of the most widely accepted protocols, namely AODV, which is one of the first ad hoc routing algorithms chosen by IETF as an experimental RFC standard [27]. While having low processing and memory overheads, AODV offers quick adaptation to dynamic link conditions.

The use of different simulation tools and their effect on the performance evaluation and comparison of different algorithms is another challenge. OpNet [4,28], NS-2 [3] and GlomoSim [2] are the most commonly used simulation tools for ad hoc network simulation. These tools have rich radio propagation, mobility, networking and routing libraries. However, these tools cannot capture the spatial aspects of the models. Some extensions, like AnSim [29] combine OpNet and GlomoSim models with a graphical interface, improving visualization of the physical environment. Nevertheless, none of these models are easy to combine with other physical models (for instance, detailed models of urban traffic, weather, natural catastrophes, etc.). Cell-DEVS, instead, enables combining different sub-models easily. Finally, in building these models as Cell-DEVS, we can make use of the existing infrastructure, including parallel simulators and distributed environments.

Cell-DEVS [8] was defined as a combination of DEVS [9] and Cellular Automata combined with timing delay functions. The DEVS formalism provides a framework for the construction of modular hierarchical models, allowing for model reuse and reducing development time and testing. In DEVS, basic models (called **atomic**) are specified as black boxes, and several DEVS models can be integrated together forming a hierarchical structural model (called **coupled**). DEVS not only proposes a framework for model construction, but also defines an abstract simulation mechanism that is independent of the model itself. A DEVS atomic is model defined as

$$AM = \langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \lambda, \text{ta} \rangle$$

With X the set of *external* events, Y the set of *output* events, S the set of *sequential* states, $\delta_{\text{ext}} : Q \times X \to S$ the *external state transition function*, where $Q := \{(s, e) | s \in S,\ 0 \leqslant e \leqslant \text{ta}(s)\}$ and e is the elapsed time since the last

state transition, $\delta_{\text{int}} : S \to S$ the *internal state transition function*; $\lambda : S \to Y$ the *output function and ta*: $S \to R_0^+ \cup \infty$ the *time advance function*. At any given time, a DEVS model is in a state $s \in S$ and in the absence of external events, it will remain in that state for a period of time as defined by ta(s). The ta(s) function can take any real value between 0 and ∞. Transitions that occur due to the expiration of ta(s) are called internal transitions. When an internal transition takes place, the system outputs the value $\lambda(s)$, and changes to state $\delta_{\text{int}}(s)$. A state transition can also happen when an external event occurs. In this case, the new state is given by δ_{ext} based on the input value, the current state and the elapsed time.

A DEVS coupled model is defined as

$$CM = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

where D is a set of components; for each $i \in D$, M_i is a component with the constraint that M_i is a DEVS model for each $i \in D \cup \{self\}$, I_i is the set of influences of i, for each $j \in I_i Z_{i,j}$ is a function, the *i-to-j* output–input translation I_i is a subset of $D \cup \{self\}$, i is not in I_i, $Z_{self,j} : X_{self} \to X_j$; $Z_{i,self} : Y_i \to Y_{self}$; $Z_{i,j} : Y_i \to X_j$ and s*elect*: subset of $D \to D$ such that for any non-empty subset E, *select* $(E) \in E$. A coupled model can have its own input and output events, as defined by the X_{self} and Y_{self} sets. Upon receiving an external event, the coupled model has to redirect the input to one or more of its components. In addition, when a component produces an output, it has to be mapped as another component's input or as an output of the coupled model itself. All these input–output mappings are defined by the Z function.

In Cell-DEVS, each cell is defined as a DEVS model, and a procedure to couple cells is used to create a complete space. Delay functions allow to define complex behavior for each cell, improving the definition for each of the sub-models: transport delays have anticipatory semantics (every output event is transmitted after a delay), and inertial delays have preemptive semantics (a scheduled event will not necessarily be executed). A Cell-DEVS atomic model is defined as:

$$TDC = \langle X, Y, S, E, d, \delta_{\text{int}}, \delta_{\text{ext}}, \tau, \lambda, \text{ta} \rangle$$

where X is a set of external input events; Y a set of external output events; S is the set of sequential states for the cell; E the set of states for the input events; d the delay function for the cell; δ_{int} is the internal transition function; δ_{ext} is the external transition function; τ is the local computing function; λ is the output function; and ta is the state's duration function. A cell uses a set of input values E to compute its future state, which is obtained by applying the local computation function τ. A delay function is associated with each cell, deferring the output of the new state to the neighbor cells. There are two types of delays: inertial and transport delays. When a transport delay is used, the future value will be added to a queue sorted by output time. Therefore, all previous values that were scheduled for output but that have not yet been sent, will be kept. On the contrary, inertial delays use a preemptive policy: any previous scheduled output value, unless the same as the new computed one, will be deleted and the new one will be scheduled. This activation of the local computation is carried by the δ_{ext} function.

After the basic behavior for a cell is defined, the complete cell space will be constructed by building a coupled Cell-DEVS model:

$$GCC = \langle Xlist, Ylist, X, Y, n, \{t_1, \ldots, t_n\}, N, C, B, Z \rangle$$

where *Xlist* is the input coupling list; *Ylist* the output coupling list; X the set of external input events; Y the set of external output events; n the dimension of the cell space; $\{t_1, \ldots, t_n\}$ the number of cells in each of the dimensions; N is the neighborhood set; C is the cell space; B is the set of border cells; and Z the translation function. This specification defines a coupled model composed of an array of atomic cells. Each cell is connected to the cells defined in the neighborhood, but as the cell space is finite, either the borders are provided with a different neighborhood than the rest of the space, or they are ''wrapped'', meaning that cells in one border are connected with those in the opposite one. Finally, the Z function defines the internal and external coupling of cells in the model. This function translates the outputs of $m - e$th output port in cell C_{ij} into values for the $m - e$th input port of cell C_{kl}. Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood.

CD++ [9] is a modeling tool that was defined using DEVS and Cell-DEVS specifications. CD++ makes use of the independence between modeling and simulation provided by DEVS, and different simulation engines

have been defined for the platform: a stand-alone version, a Real-Time simulator, and a Parallel simulator. DEVS Atomic models can be programmed and incorporated onto a class hierarchy programmed in C++. A new atomic model is created as a new class that inherits from the *Atomic* base class. The state of a model is defined in the *AtomicState* class.

The *Atomic* abstract class defines some service functions: **nextChange/lastChange** return the time until/from the next/last event; **holdIn** defines DEVS ta function; **passivate** sets the next internal transition time to infinity (the model will only be activated again if an external event is received); **getCurrentState** returns the current model's phase; **sendOutput** sends an output message through the specified *port*. A newly defined atomic model should override the following methods: **initFunction**, invoked at the first activation of the model; **externalFunction**, the δ_{ext} function of the DEVS; **internalFunction**, which defines the δ_{int} function; and **outputFunction**, the DEVS λ function. Once an atomic model is defined, it can be combined with others into a coupled model, as in Fig. 2.

This figure shows how to define one of the components of *Router* model (to be introduced later). After a model's name is defined, a list of sub-components (either an instance of an atomic model or another component) is defined using the *components* keyword. Then, a list of input and output ports is defined for the model, using the keywords *in* and *out* respectively. Once the models' ports are defined, their coupling can be described using the *link* keyword, followed by the output port for the event, and the input port that will receive it.

Cell-DEVS models are described using a built-in language provided by CD++. The model specification includes the definition of the size and dimension of the cell space, the shape of the neighborhood and borders. The cell's local computing function is defined using a set of rules with the form: POSTCONDITION DELAY {PRECONDITION}. These indicate that when the *PRECONDITION* is satisfied, the state of the cell will change to the designated *POSTCONDITION*, whose computed value will be transmitted to other components after consuming the *DELAY*. If the precondition is *false*, the next rule in the list is evaluated.

CD++ Modeler permits visualizing simulation results for 2D Cell-DEVS models in one plane, and 3D models are showed by displaying the values of all of the planes comprising the model simultaneously. Likewise, a 3D visualization GUI was built to analyze simulation results in a 3D environment [12]. In this application, the results are represented by nodes in a VRML scene. The user can navigate in the scene, and edit the nodes for more convenient analysis (Fig. 3).

3. Modeling TCP/IP in DEVS

In [30] we presented the definition of a CD++ library to simulate user-defined topologies to assess network functionality; modular design allows the addition of new models easily, while the models themselves are flexible to permit future enhancements. The library consists of two major units: data generators and inter-net-

```
class Atomic : public Model  {
public:
virtual ~Atomic();     // Destructor

protected:
//Kernel services
Time nextChange();
Time lastChange();
holdIn(AtomicState::State &, Time &);
passivate();
ModelState* getCurrentState() ;
sendOutput(Time &time,Port &port,Value value);

//User defined functions.
initFunction();
externalFunction(ExternalMessage & );
internalFunction(InternalMessage & );
outputFunction(CollectMessage & );
string className() const
};      // class Atomic
```

Fig. 1. The atomic class in CD++.

```
components : router_out@RouterOutput
out   : out
in    : from_RPU interfaceNum
link : out@router_out out
link : interfaceNum interfaceNum@router_out
link : from_RPU from_RPU@router_out

[router_out]
preparation : 050
```

Fig. 2. RouterOut coupled model.

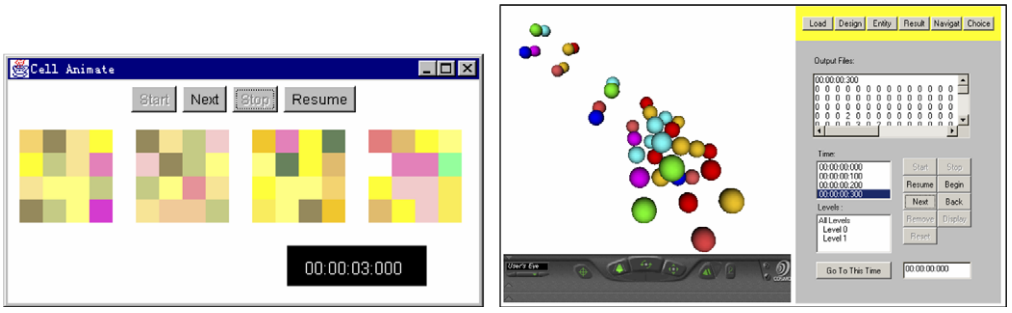

Fig. 3. CD++ Modeler and CD++ VRML GUI visualization tools.

working devices. Data generators (*host* model) were based on emulation of the TCP/IP protocol stack. Internetworking devices models include a *router* and a *hub*.

The Host coupled model is comprised of distinct models representing the Application, Transport, Network, Data Link, and physical layers. The structure of the coupled model is shown in Fig. 4. As we can see in this figure, the TCP model was broken in two (to facilitate full-duplex communications). The Transmitter module

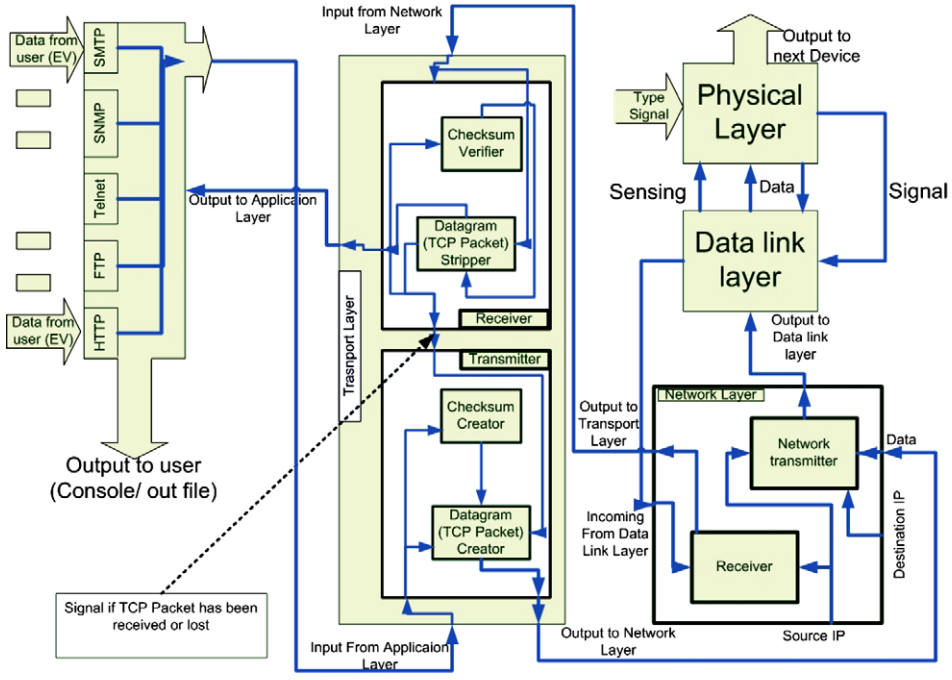

Fig. 4. Host coupled model [30].

is responsible for receiving data from the Application layer model, adding sequence and acknowledgment numbers, a window size and a checksum to the original data received (fields required for Service Level Agreement simulations).

The data received is transformed to the format shown in Fig. 5, which follows the protocol requirements [30].

Packet creation is split between two atomic models; *datagramCreator* and *checksumCreator*. Data received (from the Application layer) is routed to the *datagramCreator*, which will create an initial packet and forward it to the *checksumCreator* to compute a checksum. Then, the completed packet will be forwarded to the *datagramCreator*, and sent to the next layer in the protocol stack. Before the packet is sent, a copy is saved to accommodate the connection manager, which will resend packets in case they are not received. Each of the models was formally specified, as follows:

$$\texttt{datagramCreator} = \langle X, S, Y, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, D \rangle$$

$X = \{$*in*: receives data; *Checkin*: receives packets after the checksum has been created; *ackPort*: receives acknowledgments; *ackSender*: receives requests to send ACKs (the data received)$\}$;
$S = \{$phase, packet, saved packet, delay$\}$;
$Y = \{$*gocheck*: sends data packets to request the creation of a checksum; *datagramCreator*: sends complete output data packets; *resend* used to retransmit packets$\}$;
$\delta_{\text{int}}(s, e)$:
Case phase
  active: **passivate**;
$\delta_{\text{ext}}(s, e, x)$:
  Case msg.port
    In: Create packet;
Checkin: packet received, checksum added
ackPort: check acknowledgement
  correct?: delete saved packet
  else: resend saved packet
ackSender: send received data as ACK.
phase = active; **holdIn**(delay);
λs:
If message = packet **and** no checksum yet
  Send packet through *gocheck* (checksum = 0)
If message = data **and** checksum created
  Send data on datagramCreator
If message = ack
    Check ACK to be correct or not.
Incorrect? Discard ack; *resend* packet.
If message = request to send ack
  Send message on *resend*.

On the receiver side, we created a model to receive data from the Network layer. The model is made of two atomic components: a *datagramStripper* and a *checksumValidator*. The *datagramStripper* receives data and forwards it to the *checksumValidator* to test the checksum. If valid, the *datagramStripper* checks the packet type (data or ack). If it is data, the headers are stripped, the data is forwarded,

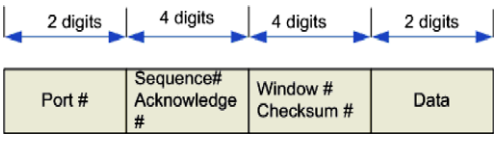

Fig. 5. TCP packet format.

and a request to the *datagramCreator* to send an ack to the source of the packet is issued. On the other hand, if the data is an ack, the *datagramStripper* forwards it to the *datagramCreator* to check if the ack is expected (either deleting the saved packet or resending it). If the checksum is incorrect, the packet is discarded.

After careful study of the model's specifications, every model was coded using the CD++ services presented in Fig. 1. Models were individually tested, and coupled models were created using the notation presented in Fig. 2. Finally, integration testing was carried out. Fig. 6 shows a detailed execution log of the Transport Layer model, in which we present the data being manipulated by its various models.

The first event (X) is an input carrying the value 12 through the HTTP port 80. This is transmitted to *datagramcreator*, which executes the external transition function (adding the window size, acknowledgment, and sequence number to the data – for testing purposes, the values = 0). Then, it schedules an internal transition (D) in 5 ms (reflecting the delay of the circuit). When this time is consumed, an internal transition (*) is fired. The first step involves executing the output function (Y), which transmits the packet through the *gocheck* port. The model then passivates ("..." represents time = ∞). This event is converted into an input (X) for *checksumcreator*, which receives the Application data and computes the checksum (also taking 5 ms). Once the checksum is computed, it is sent to the *datagramcreator* to signal that it is ready to be sent.

The data presented on the previous example arrived to the Transport layer through the higher level Application layer. This layer models a host generating and receiving data routed from various inter-networking devices, and different services and protocols. The data generated is depicted in Fig. 7. The Application layer receiving this data adds Application port variable, and then outputs the information to the Transport layer (for instance, the HTTP request generated at 09:500 is the one originating the sequence presented in Fig. 6).

```
X/00:10:000/top/in/1280 to datagramcreator
D/00:10:000/datagramcreator/005 to top
*/00:10:005/top to datagramcreator
Y/00:10:005/datagramcreator/gocheck/1200000000080 to top
D/00:10:005/datagramcreator/... to top
X/00:10:005/top/in/1200000000080 to checksumcreator
D/00:10:005/checksumcreator/005 to top
*/00:10:010/top to checksumcreator
Y/00:10:010/checksumcreator/checksumcreatorout/1200000009280 to top
D/00:10:010/checksumcreator/... to top
X/00:10:010/top/checkin/1200000009280 to datagramcreator
D/00:10:010/datagramcreator/005 to top
*/00:10:015/top to datagramcreator
Y/00:10:015/datagramcreator/datagramcreatorout/1200000009280 to top
...
```

Fig. 6. Transport layer log file.

```
INPUT
// data input on HTTP input port
00:07:000 infromHTTPuser 11
00:09:500 infromHTTPuser 12
...
// data input on Port 25
03:00:00 infromSMTPuser 13
03:10:00 infromSMTPuser 14

OUTPUT
//Application data sent on HTTP Port
00:07:500 outtoTransport 1180
00:10:000 outtoTransport 1280
...
// Application data sent on Port 25
03:00:480 outtoTransport 1325
03:10:530 outtoTransport 1425
```

Fig. 7. Application output file.

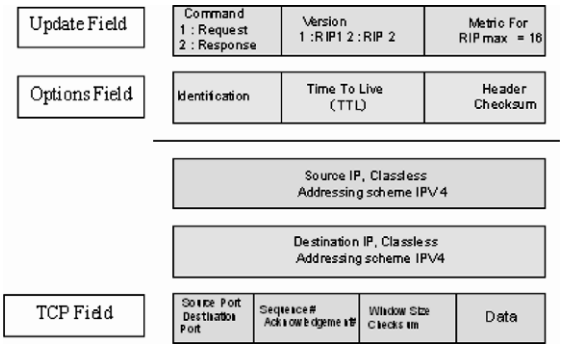

Fig. 8. Header format.

The Network layer is usually where most of the delay and stochastic operation occurs due to the nature of IP being a connectionless protocol. The layer adds a source and destination IP fields to the packet to enable routing, creating subnets, local networks, and many other Network artifacts, as shown in Fig. 8.

The headers for the Internet Protocol are based on RFC # 791 [30]. They contain the full addressing information (source and destination IP) as well as other Quality of Service parameters such as Time To Live (TTL), identification, and a checksum. The traffic packets are made of four values: the source address, the destination address, and the TCP field. The options in each field are chosen from the IPV4 packet format. As seen in Fig. 4, the Network model consists of a transmitter and a receiver, which add/extract the corresponding information using the header format in Fig. 8. Fig. 9 shows an input example for this model.

The information sent to the Network layer is used to create a checksum value, which is used to verify the data sent over the network. The model outputs the required four fields, as in Fig. 10.

The Data Link layer in our model implements the CRC operations of the Logical Link Control (LLC) sub layer (which calculates a frame, checks the sequence, and uses it to detect errors when a frame is received), and the Carrier Sense Multiple Access with collision detection (CSMA) algorithm in the Medium Access Control (MAC) sub layer, which would senses the carrier by sending a `senseCarrier` port message to the Physical layer, and waits for a response.

The Physical layer model simulates the wiring connecting different devices, using a list to save the incoming data, and outputting them at a specific time intervals, modeling the link delays. As seen in Fig. 4, the model

```
010 infromTransport 1122334455580  // data
020 DestinationIP 192168111223     // IP value
```

Fig. 9. IP test values.

```
Y/00:13:020/netwXmit1/out/4850000155000 to top
Y/00:13:020/netwXmit1/out/1921681162240 to top
Y/00:13:020/netwXmit1/out/1921681162240 to top
Y/00:13:020/netwXmit1/out/122233343180800 to top
```

Fig. 10. Network layer log file.

```
00:10:00 FTP_In 11
00:10:00 Destination 192168111
00:10:01 statusCarrier 1
00:40:02 FTP_In 1001214
00:40:02 Destination 192168001
00:40:03 statusCarrier 1
00:80:04 FTP_In 1001215
00:80:04 Destination 192168001
00:80:06 statusCarrier 1
01:90:07 Telnet_In 1001216
01:90:07 Destination 192168001
01:90:11 statusCarrier 1
```

Fig. 11. Integration test suite.

```
Y/49:010/netwXmit1/out/2000000000 to top
Y/49:010/netwXmit1/out/111222333 to top
D/49:010/netwXmit1/... to top
Y/49:010/top/outtoData Link/2000000000 to Root
Y/49:010/top/outtoData Link/111222333 to Root
```

Fig. 12. Host log file section.

interacts with others through the Data link Layer and the `sensing` port. The layer can have one of four states (`idle`, `busy`, `jammed`, `collision`) that determine how data is handled.

Data is received seamlessly through the same set of layers in the reverse direction with each layer stripping the extra variables added by its counterpart. Figs. 11 and 12 show the results of one of the integration tests for a host whose source IP address is 111222333.

The event file shows FTP data from the host to another end on the network. Simple values were chosen here, to ease the process of reviewing the results. The host reacted to these events, as shown in Fig. 12.

This section of the host log file shows events that represent the host sending the received data through the network (after adding the appropriate headers) and forwarding it to the Data Link layer. The Data Link Layer actually responded as in the following figure:

```
Y/06:000/internet/outtoData Link/ 20000 to top
Y/06:000/internet/outtoData Link/192168116224 to top
D/06:000/internet/... to top
X/06:000/top/getpacket/ 20000 to Data Link
X/06:000/top/getpacket/192168116224 to Data Link
```

The *router* model defines how to interconnect network devices. We used an abstract look up in the routing process, considering three main functionalities: receiving and forwarding traffic, processing IP packets, and maintaining a routing table.

In order to simulate the three functions, two models were created; the `RouterInterface` and the `RouterProcessor`, which in turn is made of the *ProcessingUnit* model and a *ripTable*. The router coupled model is shown in Fig. 13 (each of the models is even subdivided in lower levels of abstraction that are not discussed here). Every router has a number of interfacing cards to receive/forward traffic from/to the network. The *RouterInterface* model was developed to receive and send packets with the format discussed in Fig. 8. To handle the traffic going in/out of the router, the `RouterInterface` was designed as a coupled model consisting of one model receiving packets from the network, and a second forwarding packets out of the router. After packets are received by the `RouterInterface`, they are processed to see if they are messages to the router (requests or updates), or just data packets to be forwarded to their destinations.

The `ProcessingUnit` is responsible for reading in the packets from the interfaces, processing them, and making routing decisions regarding their destinations. Upon receiving a packet, it looks at the packet's header, extracts from it the TTL value, and checks if it is valid. If it is valid it will read the packet's type, and will react according to type.

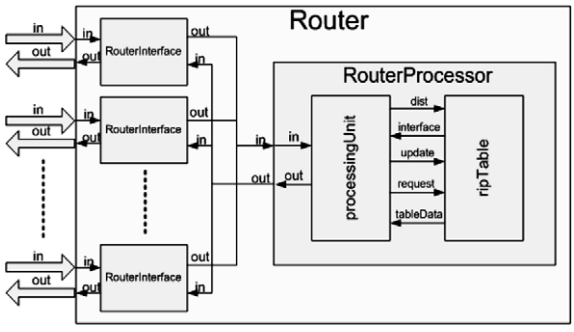

Fig. 13. Router's coupled module.

Three types of packets are accepted: *respond*, *request*, and *data*. *Request* packets carry the destination address of the requesting router that wants the update, following the RIP protocol [31]. This address value is extracted from the packet's header and it is sent along with the requesting router reply information to the *ripTable* model, so that the proper reply information can be prepared and sent to the requesting router. The *respond* packets carry a network address and a metric value (cost) associated with that route. These packets are used to update other routers, or to respond to other routers' requests for updates. The router extracts both the address and the metric, and it forwards this information along with the sending router's data to the *ripTable*. When a *data* packet is received, the *processingUnit* extracts its destination address, and forwards it to the *ripTable* model (which maintains the routing information for forwarding packets). Once the *ripTable* returns an output interface, the *RouterProcessor* will simply forward the data packet through it. If the destination address is not found in the routing table, the value 0 is returned (and a request packet is issued through all interfaces except the one that the packet was received through, requesting an update on that destination).

The *ripTable* is in charge of maintaining the routing information that the router needs to forward packets to its destinations. The entries in the table have the format <Address,*Metric*,*Interface*>. *Address* is a destination for the packet; *Metric* represents the cost of getting to that destination, and the output Interface is the one through which the router must forward the packet (in order to be at least one hop closer to the destination).

The *ripTable* receives three events: *update*, request, and *request for forwarding* information. In the case of *update*s, the model will be receiving the address that the update is about, together with a new metric value. If the address does not exist, the information will be added. Otherwise, the associated metric is compared with the newly received one, and it will replace the output interface number with that of the new update, if the new metric value is smaller than the one in the table. For the *request* events, the *ripTable* model will prepare the required information from its table, and redirects it as responds. Finally, for the *forwardinformation* request, the model will search its table for the destination address and send the output interface that should be used to forward the packet.

The router's behavior was tested using different scenarios, as shown in Fig. 14.

```
INPUTS
00:00:010 in1 2000001  // update with metric 1
00:00:010 in1 111101101  // address
...
00:00:100 in1 3010012 // data, ttl=10, CRC=12
00:00:100 in1 121117001 // source address
00:00:100 in1 133303303 // destination address
00:00:100 in1 15
00:01:010 in1 2000000   // update metric 0
00:01:010 in1 133303303
...
00:02:000 in1 3008011 // data,ttl=8, CRC = 11
00:02:000 in1 114124201
00:02:000 in1 123456789 // unknown destination 00:02:010 in2 2000007   // update metric 7
00:02:010 in2 122202202
00:02:010 in1 3000007  // data, TTL = 0
00:02:010 in1 122202202

OUTPUTS
00:00:018 out2 2000001  // update
00:00:018 out2 111101101  // address
...
00:00:109 out2 3010012 // data forward
00:00:109 out2 121117001
00:00:109 out2 133303303
00:00:109 out2 15
00:01:018 out2 2000000   // update
00:01:018 out2 133303303
...
00:02:009 out2 1000000// request
00:02:009 out2 123456789
```

Fig. 14. Router input/output events.

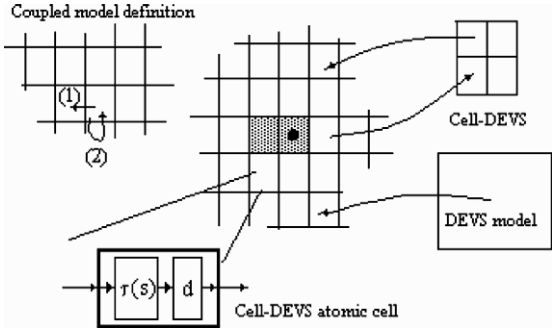

Fig. 15. Integrating DEVS (i.e., router, host) with Cell-DEVS (AODV) models.

The first packet is an update. The router passes the related values to its table and the table is updated. The message arrived at the router from interface 1, and a corresponding update message was created and sent through interface 2 (*out2*). For every update packet, an update to the neighbor nodes is sent thought the other router interface. Then, we show a packet representing data injected into the router. The packet option field shows a TTL value of 10. The router knew the address since it received an update on it before. The router forwards the packet using the right output interface. After that, another update with a smaller metric for an address that the router has in its table is sent through interface 1. We can see that the router did update its table with the better metric value and sent an update through interface 2.

No output was sent in response to the last two packets. The reason is that the first one was an update with a metric higher than the existing one in the routing table. The second was a data packet with a TTL value of 0 (expired). In both cases, the router discarded the packets.

These models of inter-networking are capable of building topologies as a first step for building a more complex library. The models chosen are sufficient to create network topologies with an acceptable level of accuracy in services, and customization in terms of Quality of Service and Service Level Agreements, and they provide the backbone for a larger model library, since all components chosen represented different fields and layers of a typical packet switched network.

4. Modeling routing in wireless ad hoc networks using Cell-DEVS

We used Cell-DEVS to model the core functionality of the Ad Hoc On-Demand Distance Vector Protocol (AODV) [12]. AODV assumes symmetric (bi-directional) links and creates routes between two nodes only ''on demand'' to eliminate overheads. Whenever, a node wants to communicate to another it broadcasts a Route Request (RREQ) message to its neighbors. The neighbor re-broadcasts the message and set up a reverse path pointing towards the source. When the intended destination receives a RREQ message, it replies by sending a Route Reply (RREP) that travels along the reverse path set up when the RREQ is forwarded.

Our model considers a network plane divided into cells where nodes are spread randomly on the plane. The network plane does not make any assumptions about the physical location of the nodes in the area. Thus, each cell in the network plane may have a different size in terms of physical area represented (a few square meters in a highly dense networks or a few square kilometers in a less dense networks). The key is, that movement of data between two cells represents one hop. The reason for modeling it in such a way is that in AODV, routing takes into account the hop count instead of the actual physical distance the data has to travel. Each node can communicate to the neighboring nodes (representing those which lie within the direct communication range). The network plane also contains *dead cells* through which communication cannot take place. These cells represent physical obstacles (such as a high-rise building) or simply the absence of a communication link. Two nodes with a dead cell between them cannot communicate directly to each other. If we assume the cost of the communication links (distance vectors) between any two nodes that can directly communicate to each other to be the same, modeling AODV using Cell-DEVS involves finding the shortest path between two nodes in the ad hoc network plane.

In an ad hoc network, nodes are spread randomly and there may be many physical obstacles in between the nodes. In a real-world scenario, often nodes that need to communicate are several hops away. Thus, for a model of ad hoc network to be representative of a real-world scenario it should take into account the above-mentioned factors. For experimental results to be presented in Sections 4.1–4.3 and 4.4, we generated test models by randomly spreading the nodes in the network plane. Dead cells which are representative of communication obstacles were also spread randomly throughout the network and we ensure that in most of the scenarios source and destination nodes are several hops away. These considerations made tests more representatives of real-world scenarios. For testing each model several factors such as location of source-destination pairs, size of the network plane, dimensions of the network plane, number of nodes in the plane and the number of communication obstacles are varied. The validity of results was confirmed by manually testing the results obtained. As CD++ Modeler provides a good visualization of the results obtained, the validation process was simple. We present snapshots of our tests taken with CD++ GUI during different steps of tests conducted and with the help of those snapshots one can easily verify the correctness of the results obtained.

4.1. Defining AODV using Cell-DEVS

As discussed, modeling AODV involves finding the shortest path between two nodes in an ad hoc network plane. In order to find out the shortest path between two communicating nodes on a network plane, we have made use of a variant of the classical Lee's Algorithm [11]. Fig. 16 shows a simple example of a network plane. Here, S represents a sender node and D a destination node while black cells represent dead cells. In order to find a route from S to D, the node S broadcasts RREQ message to all its neighbors (called the *wave* nodes). The wave nodes re-broadcast the message to their neighbors, and set up a reverse path to the sender, which is represented with pointing arrows in the figure. These nodes further re-broadcast this message and set up a reverse path to the nodes from which they received the message. This process continues until the message reaches the destination node D.

Since there are more than one path from the sender to the destination, the destination may receive multiple RREQ message for the same sender. However, the route through which the destination node receives the RREQ message first is the shortest path between the sender and the destination. The destination thus ignores all RREQ messages for the same sender except the first one. It replies to the first RREQ message sending a RREP message using the reverse path set up when the RREQ messages are forwarded. All the wave nodes that lie on this shortest route between the sender and the destination become the *path* nodes (represented with circles containing arrows in the figure). All communications between the sender and the destination from this point onwards takes place using this path until the topology of the network changes. All other wave nodes are sent a clear state message to move them from the *wave* state to a *clear* state (not shown in the figure).

This model can be formally defined using Cell-DEVS specifications as follows:

$$\mathrm{CD} = \langle X, Y, S, E, d, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \tau, \lambda, D \rangle$$

$S =$ **dead** (Dead Cell), **init** (Initial State of the Nodes), **initD** (Initial State of the Destination Node), **DR** (Destination Ready; state of the Destination Node after it has received a send request from the sender), **InitS** (Initial State of the Sender Node), **WaveU** (Wave Up ↑),**WaveD** (Wave Down ↓), **WaveR** (Wave Right →),

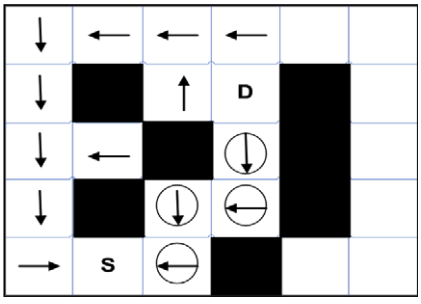

Fig. 16. AODV routing in Cell-DEVS.

WaveL (Wave Left ←), **PathU** (Path Up ↑), **PathD** (Path Down ↓), **PathR** (Path Right →), **PathL** (Path Left ←), **Clear** (final state of the node that received a wave message but is not going to become a path node), **Found** (destination found; final state of the Sender Node).

$X = \Phi$, $Y = \Phi$; $E = \{(-1,0), (0,-1), (0,0), (0,1), (1,0)\}$, $d = 100$ ms; $\tau : E \rightarrow S$ are the rules defined according to the algorithm discussed and δ_{int}, δ_{ext}, λ, D are defined according to the definitions of Cell-DEVS atomic models.

Using this specification as a base, the model was implemented in CD++, as presented in Fig. 17. Note here that in [path-rule] many Boolean statements could have been combined together. However, for the sake of clarity each statement is written in a separate line.

A number of tests were conducted on the model, and we will present here one of them as an example (more detailed results can be found in [33]). Moreover, Fig. 18 shows screen shots taken from the execution of the model on CD++ at different intervals during the test. Here all the dead cells are represented in black and all the cells that have not yet received any message in white. This example used a 20×28 cells rectangular network plane. The initial distribution of the nodes is given in Fig. 18a. The sender node (shown in gray in the lower-left part of the figure) wants to communicate to a certain destination node (shown in the top-right part of the figure). The first step is to broadcast a RREQ message as defined by rules $Wave\{U, D, L, R\}$ in Fig. 17. The state of the model after 50 steps of execution is shown in (b). Here the light gray nodes represent those nodes that have received a RREQ message and have re-broadcasted this message while establishing a reverse path to the sender. The dark gray node (on the way to the destination) is a node that carries a RREP message from the destination and is going to become a path from the sender to the destination. This is in accordance with the

```
[path]
type : cell      width : 20  height : 28   delay : transport      border : nowrapped
neighbors : (-1,0) (0,-1) (0,0) (0,1) (1,0)  localtransition : path-rule

[path-rule]
rule : DR 100 { (0,0) = InitD and stateCount(PathU) > 0}
rule : DR 100 { (0,0) = InitD and stateCount(PathD) > 0}
rule : DR 100 { (0,0) = InitD and stateCount(PathR) > 0}
rule : DR 100 { (0,0) = InitD and stateCount(PathL) > 0}
rule : WaveU 100 { (0,0) = Init and (-1,0)> DR and (-1,0)< PathU}
rule : WaveD 100 { (0,0) = Init and (1,0) > DR and (1,0) < PathU}
rule : WaveR 100 { (0,0) = Init and (0,1) > DR and (0,1) < PathU}
rule : WaveL 100 { (0,0) = Init and (0,-1)> DR and (0,-1)< PathU}
rule : PathU 100 { (0,0) = WaveU and stateCount(InitD) = 1}
rule : PathD 100 { (0,0) = WaveD and stateCount(InitD) = 1}
rule : PathR 100 { (0,0) = WaveR and stateCount(InitD) = 1}
rule : PathL 100 { (0,0) = WaveL and stateCount(InitD) = 1}
rule : PathU 100 { (0,0) = WaveU and (0,-1)= PathR}
rule : PathU 100 { (0,0) = WaveU and (0,1) = PathL}
rule : PathU 100 { (0,0) = WaveU and (1,0) = PathU}
rule : PathD 100 { (0,0) = WaveD and (0,-1)= PathR}
rule : PathD 100 { (0,0) = WaveD and (0,1) = PathL}
rule : PathD 100 { (0,0) = WaveD and (-1,0)= PathD}
rule : PathR 100 { (0,0) = WaveR and (0,-1)= PathR}
rule : PathR 100 { (0,0) = WaveR and (-1,0)= PathD}
rule : PathR 100 { (0,0) = WaveR and (1,0) = PathU}
rule : PathL 100 { (0,0) = WaveL and (0,1) = PathL}
rule : PathL 100 { (0,0) = WaveL and (-1,0)= PathD}
rule : PathL 100 { (0,0) = WaveL and (1,0) = PathU}
rule : clear 100 { (0,0) = Init and stateCount(clear) > 0}
rule : clear 100 { (0,0) > InitS and (0,0) < PathU and stateCount(clear) > 0}
rule : clear 100 { (0,0) > InitS and (0,0) < PathU and stateCount(DR) > 0}
rule : clear 100 { (0,0) > InitS and (0,0) < PathU and stateCount(found) > 0}
rule : clear 100 { (0,0) > InitS and (0,0) < PathU and (-1,0) >WaveL and (-1,0)<clear and (-1,0)!=PathD}
rule : clear 100 { (0,0) > InitS and (0,0) < PathU and (1,0) > WaveL and (1,0) < clear and (1,0)!=PathU}
rule : clear 100 { (0,0) > InitS and (0,0) < PathU and (0,-1) >WaveL and (0,-1)<clear and (0,-1)!=PathR}
rule : clear 100 { (0,0) > InitS and (0,0) < PathU and (0,1) > WaveL and (0,1) < clear and (0,1)!=PathL}
rule : found 100 { (0,0) = InitS and stateCount(PathU) > 0}
rule : found 100 { (0,0) = InitS and stateCount(PathD) > 0}
rule : found 100 { (0,0) = InitS and stateCount(PathR) > 0}
rule : found 100 { (0,0) = InitS and stateCount(PathL) > 0}
rule : {(0,0)} 100 {t}
```

Fig. 17. Implementing AODV routing in CD++ [32].

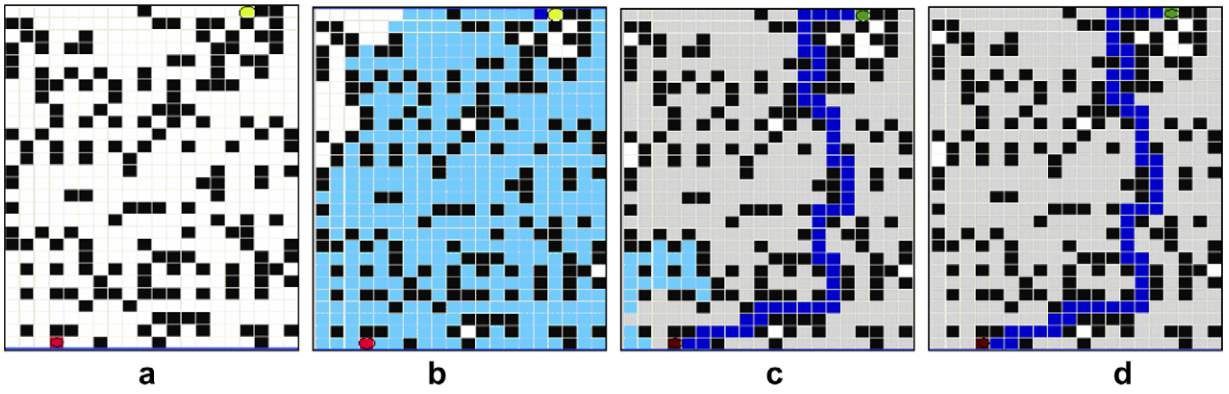

Fig. 18. AODV simulation: (a) initial distribution, (b) state after 50 steps, (c) after 100 steps and (d) final state after 106 steps.

rules $Path\{D, R, L, U\}$ in Fig. 17. The state of the model after 100 steps of execution is shown in Fig. 18c. Here a successful route has been established between the sender and the destination. Wave nodes that are not going to become path nodes have been sent a clear state message. They are represented in light gray (rule *clear* in Fig. 17). The final state of the model after 106 steps is shown in Fig. 18(d). Note that the model has successfully established the shortest route between the sender and the receiver and all wave nodes have ended up in clear state. The white nodes represent those nodes that do not receive RREQ message.

4.2. Inter-networking AODV models using 3D Cell-DEVS

We extended the above model to three dimensions, as a way of representing different networks to which the given ad hoc network connects, enabling inter-network routing. We can have different costs of communication in each network and thus it may be desirable to route the data through the network(s) which has (have) the least communication costs. For example, if one of the planes represents a wireless ad hoc network and the other a wired network, it would make sense to transmit the data in the ad hoc plane to the wired plane through the nearest gateway (because the cost of the communication in a wired network is generally less than that in a wireless one). We present here a simple case in which the intra-network and inter-network communication costs are assumed to be the same (the model, however, is extendable to the case where different costs are associated with each hop. Each *wave* node broadcasts the RREQ message and the total cost of the reverse path from this node to the sender on the reverse path. Each node that receives that RREQ message can calculate the total cost to the sender on the reverse path by adding in the total cost to the sender, the cost of communication of the link from which it received the message. The RREP message on its way from the destination to the sender always selects the *path* with the least communication costs). We can implement this model in Cell-DEVS by extending the specification discussed in Section 4.1, and adding 4 new states to the model:

$S =$ **dead** (Dead Cell), **init** (Initial State of the Nodes), **initD** (Initial State of the Destination Node), **DR** (Destination Ready; state of the Destination Node after it has received a send request from the sender), **InitS** (Initial State of the Sender Node), **WaveU** (Wave Up), **WaveD** (Wave Down), **WaveR** (Wave Right), **WaveL** (Wave Left), **Wave3P** (Wave in positive 3rd dimension) , **Wave3N** (Wave in negative 3rd dimension) , **PathU** (Path Up), **PathD** (Path Down), **PathR** (Path Right), **PathL** (Path Left), **Path3P** (Path in positive 3rd dimension), **Path3N** (Path in negative 3rd), **Clear** (final state of the node that received a wave message but is not going to become a path node), **Found** (dimension destination found; final state of the Sender Node).

Using these states, the algorithm for two-dimensional plane was extended to three dimensions. The algorithm works the same way as the algorithm in Section 4.1 with a difference that the new algorithm also takes into account the 3rd dimension while broadcasting RREQ messages and developing path from the destination to the sender. The implementation of the new model along with the set of rules is shown in Fig. 19.

A number of tests were conducted on this model (more detailed results can be found in [33]), and here we present the results obtained in an example consisting of a $5 \times 5 \times 5$ cell space. Like before, all the dead cells are

```
[path]
type : cell    dim : (5,5,5)      delay : transportdefaultDelayTime : 100      border : nowrapped
neighbors : (0,0,1)(-1,0,0)(0,-1,0)(0,0,0)(0,1,0) (1,0,0) (0,0,-1)
localtransition : path-rule

[path-rule]
rule : DR 100 { (0,0,0) = initD and (stateCount(Path3P) > 0 or stateCount(Path3N)> 0 or stateCount(PathU) > 0 or state-
Count(PathD) > 0 or stateCount(PathR) > 0  or stateCount(PathL) > 0)}
rule : WaveU 100 { (0,0,0) = init and (-1,0,0) > DR and (-1,0,0) < PathU}
rule : WaveD 100 { (0,0,0) = init and (1,0,0) > DR and (1,0,0) < PathU}
rule : WaveR 100 { (0,0,0) = init and (0,1,0) > DR and (0,1,0) < PathU}
rule : WaveL 100 { (0,0,0) = init and (0,-1,0) > DR and (0,-1,0) < PathU}
rule : Wave3P 100 { (0,0,0) = init and (0,0,1) > DR and (0,0,1) < PathU}
rule : Wave3N 100 { (0,0,0) = init and (0,0,-1) > DR and (0,0,-1) < PathU}
rule : PathU 100 { (0,0,0) = WaveU and stateCount(initD) = 1}
rule : PathD 100 { (0,0,0) = WaveD and stateCount(initD) = 1}
rule : PathR 100 { (0,0,0) = WaveR and stateCount(initD) = 1}
rule : PathL 100 { (0,0,0) = WaveL and stateCount(initD) = 1}
rule : Path3P 100 { (0,0,0) = Wave3P and stateCount(initD) = 1}
rule : Path3N 100 { (0,0,0) = Wave3N and stateCount(initD)= 1}
rule : PathU 100 { (0,0,0) = WaveU and (0,-1,0) = PathR}
rule : PathU 100 { (0,0,0) = WaveU and (0,1,0) = PathL}
rule : PathU 100 { (0,0,0) = WaveU and (1,0,0) = PathU}
rule : PathU 100 { (0,0,0) = WaveU and (0,0,1) = Path3N}
rule : PathU 100 { (0,0,0) = WaveU and (0,0,-1) = Path3P}
rule : PathD 100 { (0,0,0) = WaveD and (0,-1,0) = PathR}
rule : PathD 100 { (0,0,0) = WaveD and (0,1,0) = PathL}
...
rule : Path3N 100 { (0,0,0) = Wave3N and (0,0,1) = Path3N}
rule : clear 100 { (0,0,0) = init and stateCount(clear)>0}
rule : clear 100 { (0,0,0) > InitS and (0,0,0) < PathU and stateCount(clear) > 0}
rule : clear 100 { (0,0,0) > InitS and (0,0,0) < PathU and stateCount(DR) > 0}
rule : clear 100 { (0,0,0) > InitS and (0,0,0) < PathU and stateCount(found) > 0}
...
rule : found 100 { (0,0,0) = InitS and stateCount(PathR) >0}
rule : found 100 { (0,0,0) = InitS and stateCount(PathL) >0}
rule : found 100 { (0,0,0) = InitS and stateCount(Path3P) >0}
rule : found 100 { (0,0,0) = InitS and stateCount(Path3N) >0}
rule : {(0,0,0)} 100 {t}
```

Fig. 19. Implementing 3D AODV routing in CD++ [32].

represented in black and all the cells that have not yet received any message in white. The initial distribution of the nodes is given in Fig. 20a. The sender node near the bottom of the figure broadcasts the RREQ message that now is transmitted in three dimensions. The state of the model after 10 steps of execution is shown in Fig. 20b. Here the destination node near the top has received the RREQ message and has sent a RREP message. The path is thus being formed between the sender and the destination represented in dark gray. The final state of the model after 18 steps of executions is shown in Fig. 20c. Note that the model has successfully established the shortest path between the sender and the receiver in a three-dimensional space.

4.3. Modeling multicast AODV

We extended the model in Section 4.1 to model multicasting in AODV. The construction of multicast trees on cellular planes is a complex task, and Hochberger [23] has shown that as the number of receiver and/or

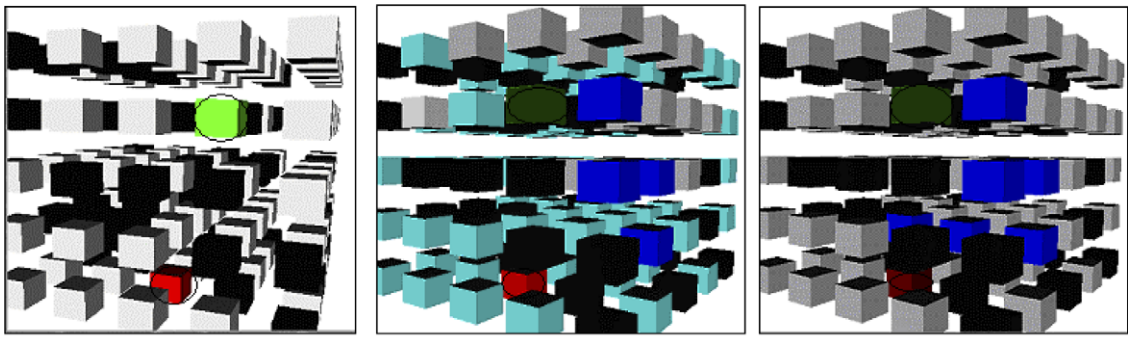

Fig. 20. 3D AODV simulation: (a) initial distribution, (b) state after 10 steps and (c) after 18 steps.

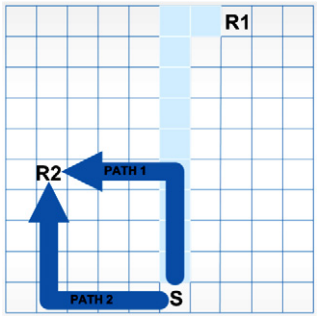

Fig. 21. Optimal multicast trees concept.

sender nodes on a plane increases, the number of states for each cell goes beyond practical limits. We introduce a new algorithm to overcome this problem, and by introducing the notion of trees, model AODV multicast with a small number of states for each cell. Moreover, in our implementation, the number of states for each cell is independent of the number of nodes in the tree (for the sake of simplicity we have not modeled the group leader and the sequence numbers in standard multicast AODV).

Another aspect that has been taken into account during the construction of multicast trees is its optimality, i.e., the multicast tree should duplicate data as less as possible. Consider for example the distribution of nodes shown in Fig. 21. Here S represents a sender node that wants to multicast data to two nodes R1 and R2. The shortest path between S and R1 exists and is shown in the figure in light gray. Note that for R2, there are two shortest paths, both of which involve 8 hops from S to R2. *Path 1* takes data first to the top of S and then left to R2 while *Path 2* takes data first to the left of S and then top to R2. Note that as the data is being multicast by S to the two nodes R1 and R2, it would make much more sense to send data to R2 through *Path 1*. This is because this way data would be duplicated only for the last 4 hops from S to R2. On the other hand, if the data is sent to R2 through Path 2 it is duplicated right in the beginning (i.e., a total of 8 hops). If there are multiple nodes in a multicast group, less duplicating data would save bandwidth. The construction of multicast trees in our model considers this feature and seeks to minimize data duplication.

Our proposed algorithm for modeling Multicast AODV can be summarized in the following steps:

1. The model establishes the shortest route between the sender and one receiver using the algorithm discussed in Section 4.1.
2. All the path nodes from the sender to the receiver become *tree* nodes, and a tree is formed between the sender and the receiver. Note that the sender and a receiver belong to the multicast group but the tree nodes may or may not be a part of the group. During this step all clear state nodes are also re-initialized to their initial state value.
3. A new node wanting to join the multicast group broadcasts a RREQ message to join the group.
4. A path is established between the new node and the nearest tree node, following the algorithm in step 1 (instead of a particular receiver node, the path is formed between the new node and the nearest tree node).
5. Since the wave messages are broadcasted in fourth step and there may be more than one tree nodes in the model, step 4 may generate more than one path between the tree and the new node. During step 5, all such undesirable paths are purged by sending out a clear state message. This is done by adding a logic to the model to detect and purge non-optimal paths.
6. The successful completion of step 5 generates the shortest, optimal path from the new node to the tree. This path becomes the tree during this step. Moreover, during this phase, all clear state nodes are re-initialized to their initial value.
7. For each subsequent node that wants to join the multicast group, steps 3–6 are repeated. The algorithm, thus, can add as many nodes to the multicast trees as desired.

The formal specifications of this model are similar to the one presented in Section 4.1. There are two major differences: the definition of the cell's updating rules, which are now according to the algorithm just defined,

```
components : path          in : in1 in2 in3
link : in1 in1@path                link : in2 in2@path                link : in3 in3@path

[path]
type : cell                width : 20       height : 28                delay : transport
in : in1 in2 in3 border : nowrapped
link : in1 inp@path(9,3)
link : in2 inp@path(16,19)
link : in3 inp@path(23,19)
neighbors : (-1,0)   (0,-1)  (0,0)  (0,1) (1,0)
localtransition : path-rule
portInTransition : inp@path(9,3) special-rule
portInTransition : inp@path(16,19) special-rule
portInTransition : inp@path(23,19) special-rule
[path-rule]
rule : DR 100 { (0,0) = initD and stateCount(PathU) > 0}
rule : DR 100 { (0,0) = initD and stateCount(PathD) > 0}
rule : DR 100 { (0,0) = initD and stateCount(PathR) > 0}
rule : DR 100 { (0,0) = initD and stateCount(PathL) > 0}
rule : WaveU 100 { (0,0) = init and (-1,0) > DR and (-1,0)< PathU}
rule : WaveD 100 { (0,0) = init and (1,0) > DR and (1,0) < PathU}
rule : WaveR 100 { (0,0) = init and (0,1) > DR and (0,1) < PathU}
rule : WaveL 100 { (0,0) = init and (0,-1) > DR and (0,-1)< PathU}
rule : PathU 100 { (0,0) = WaveU and (stateCount(initD) = 1 or stateCount(tree) = 1) and (stateCount(PathU) + state-
Count(PathD)+ stateCount(PathR)+ stateCount(PathL)= 0)}
rule : PathD 100 { (0,0) = WaveD and (stateCount(initD) = 1 or stateCount(tree) = 1) and (stateCount(PathU) + state-
Count(PathD)+ stateCount(PathR)+ stateCount(PathL)= 0)}
rule : PathR 100 { (0,0) = WaveR and (stateCount(initD) = 1 or stateCount(tree) = 1) and (stateCount(PathU) + state-
Count(PathD)+ stateCount(PathR)+ stateCount(PathL) = 0)}
rule : PathL 100 { (0,0) = WaveL and (stateCount(initD) = 1 or stateCount(tree) = 1) and (stateCount(PathU) + state-
Count(PathD)+ stateCount(PathR)+ stateCount(PathL)= 0)}
...
rule : clear 100 { (0,0) > initS and (0,0) < PathU and stateCount(clear) > 0}
rule : clear 100 { (0,0) > initS and (0,0) < PathU and stateCount(DR) > 0}
rule : clear 100 { (0,0) > initS and (0,0) < PathU and stateCount(found) > 0}
rule : clear 100 { (0,0)>initS and (0,0)<PathU and (-1,0)>WaveL and (-1,0) < clear and (-1,0) != PathD}
rule : clear 100 { (0,0)>initS and (0,0) < PathU and (1,0) > WaveL and (1,0) < clear and (1,0) != PathU}
rule : clear 100 { (0,0)>initS and (0,0)<PathU and (0,-1)>WaveL and (0,-1) < clear and (0,-1) != PathR}
...
rule : clear 100 { (0,0) = PathU and (-1,0) = clear }
rule : clear 100 { (0,0) = PathD and (1,0) = clear }
rule : clear 100 { (0,0) = PathR and (0,1) = clear }
rule : clear 100 { (0,0) = PathL and (0,-1) = clear }
...
rule : MS 100 { (0,0) = found }
rule : MR 100 { (0,0) = DR and stateCount(tree) > 0 }
rule : {(0,0)} 100 {t}

[special-rule]
rule : {portValue(thisPort)} 100 {t}
```

Fig. 22. Implementing multicast AODV routing in CD++.

and the addition of three new states: **Tree** (State of the nodes that belong to the multicast tree), **MS** (Final state of the sender node after the construction of the tree), **MR** (Final state of the receiver node after the construction of the tree). The implementation of this model in CD++ is shown next in Fig. 22. The width and height of the model given here are for a particular example. Different tests were run with different values of width and height. Moreover, the events receiving nodes here are for a particular test. For different tests, different nodes generated requests to become a part of the multicast group.

Fig. 23 shows the screen shots taken from execution of a sample model at different intervals during the test. The test consists of 25×25 cell plane. Initially there are only two nodes; one at the top of the figure the other at the bottom. Subsequent nodes join the group after the successful completion of tree construction. The initial distribution of the nodes is shown in Fig. 23a. The state of the model after 50 steps of execution is shown in Fig. 23b. Here the path between the two nodes is being formed in accordance with the $path\{U, D, R, L\}$ rules in Fig. 22. Also, note that all the wave nodes are being set to clear state and the clear state nodes are being re-initialized in accordance with the rules *clear* and *init* in Fig. 22 respectively.

After 80 steps of execution, the state of the model is shown in Fig. 23c. Here the path between the two nodes has been established and that path is becoming a tree according to rule *tree* in Fig. 22. All clear state nodes have been re-initialized. The state of the model after 93 steps of execution is shown in Fig. 23d. Here the tree between the two nodes has been established and a new node near the right center of the figure has just decided to join the multicast group. The state of the model after 125 steps of execution is shown in Fig. 23e where the new node has broadcasted RREQ message. As there are multiple tree nodes, multiple RREP have been sent

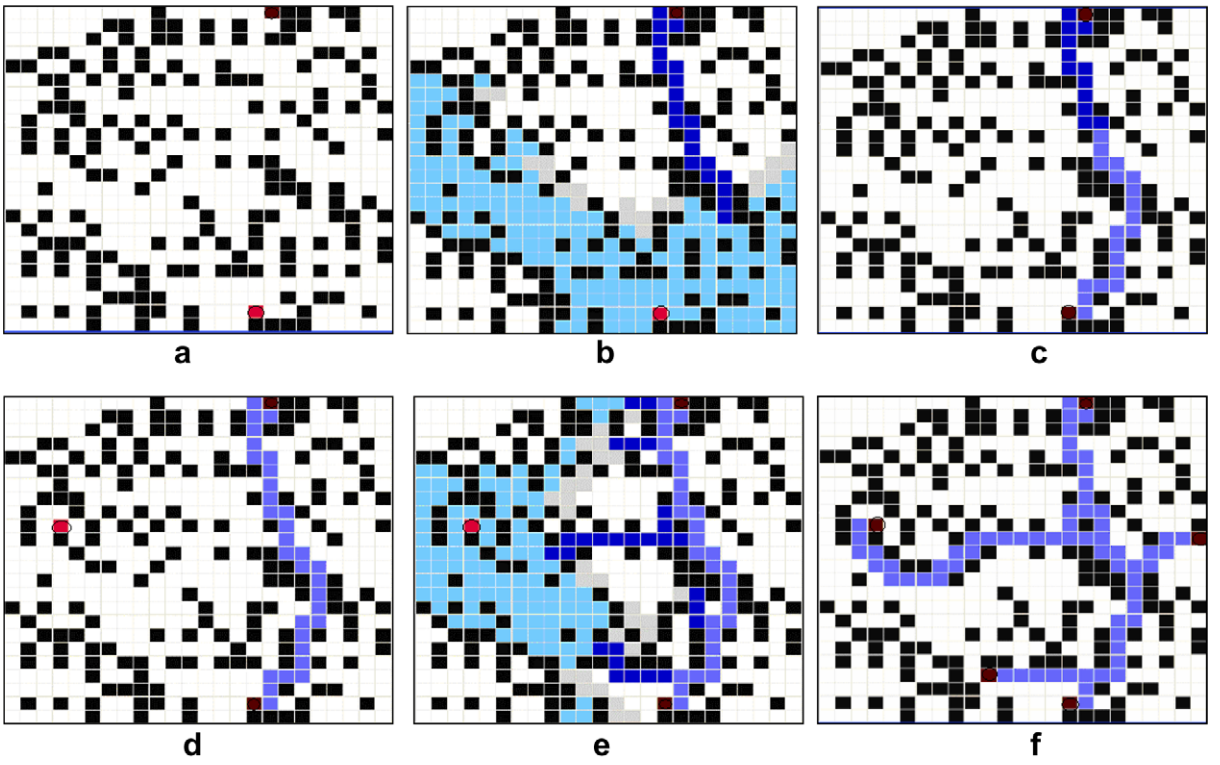

Fig. 23. AODV multicast modeling: (a) initial distribution, (b) state after 50 steps, (c) after 80 steps, (d) after 93 steps, (e) after 125 steps and (f) final state after 217 steps.

and hence multiple (non-optimal) paths from the trees are being formed. However, the algorithm logic is detecting and purging all such non-optimal paths (rule *clear* in Fig. 22). The new node find the shortest optimal route and that route becomes a tree (rule *tree*). All other non-optimal paths are purged and all the clear state nodes have been re-initialized. The final state of the model after 217 steps of execution is shown in Fig. 23f. Here 3 new nodes have been successfully added to the multicast tree formed between the first two nodes.

The above example thus shows that the model has successfully built an optimal multicast tree. Note that many non-optimal paths are generated during the addition of every new node to the group but the model successfully detects and builds the shortest path and purges all other non-optimal paths.

4.4. Modeling routing among multiple pairs of senders and receivers

We have also defined models for AODV routing among multiple pairs of senders and receivers. The variant of Lee's Algorithm used in Section 4.1 fails for multiple pairs of senders and receivers: it generates deadlocks and may prevent the generation of routing path between pairs of nodes that can communicate [23]. To overcome this problem, we have exploited the inherent parallelism in Cell-DEVS and have found a very simple solution to the problem. The idea is that each pair of sender and receiver is allocated a plane in a 3D Cell-DEVS. The total number of planes in the resulting three-dimensional model thus depends on the total number of pairs of receivers and senders to be routed in parallel. In each plane, we run the variant of Lee's Algorithm discussed in Section 4.1 (with a total of 15 states for each cell). The scheme permits to route multiple pairs of senders and receivers without having to define more states. The reason is that as each pair is routed separately in each plane, routing messages for each pair do not interfere with each other. The reason for the failure of other approaches as in [23] was that the routing messages for different pairs interfere with each other. By avoiding this interference, we can successfully prevent the generation of deadlocks. Moreover, our approach exploits the inherent parallelism in the Cell-DEVS models as multiple pairs are routed simultaneously.

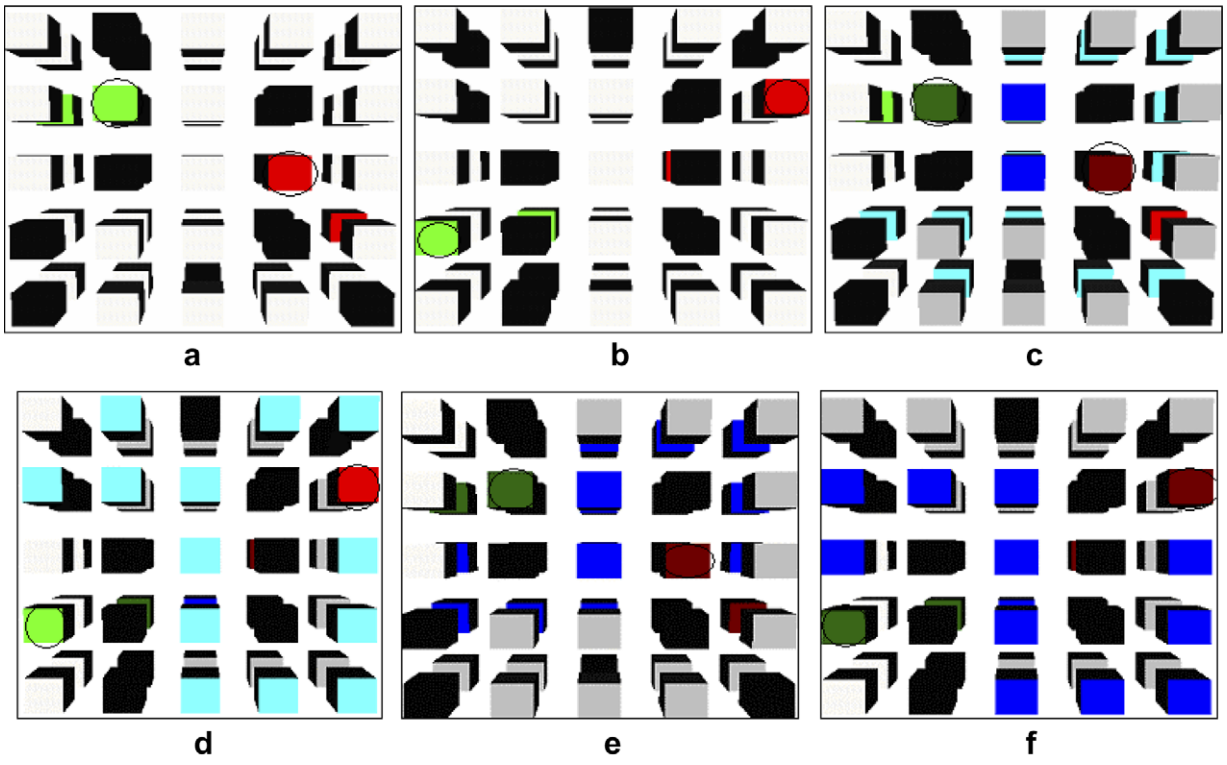

Fig. 24. AODV routing among multiple pairs of senders and receivers: (a) initial distribution of the nodes in Plane 1, (b) in Plane 2, (c,d) state after 10 steps and (e,f) final state after 25 steps.

Using the same state definitions as in Section 4.2 and similar implementation in CD++, we successfully implemented the model. Although a number of tests were conducted on the model, we present here a simple test as a sample, in order to facilitate the understanding of the execution results obtained. The test consists of a 5×5 cells space having 2 planes and thus routes two pairs of senders and receivers, one in each plane. Since the two planes represent the same network, we can assume the distribution of the nodes on the two planes to be the same. However, to show that our model works even if we assume different distribution of nodes in each plane, the example chosen has different distribution of nodes in plane 1 and plane 2 as shown in Fig. 24a and Fig. 24b respectively. The state of the model after 10 steps of execution in plane 1 and plane 2 is shown in Fig. 24c and d respectively. In plane 1, a successful route has been established between the sender and the receiver while in plane 2 RREQ message has not yet reached the destination. The final state of the model in plane 1 and plane 2 after 25 steps of execution is shown in Fig. 24e and f respectively. The figures show that the model has successfully established the shortest path between the sender and the receiver in each of the planes. The model thus is capable of routing among multiple pairs of senders and receivers simultaneously without having to define more states. Although we have shown an example in which two pairs are routed simultaneously, an arbitrary number of pairs can be routed simultaneously by defining each pair in a separate plane.

5. Modeling the behavior of mobile nodes

We model the mobility behavior of mobile nodes for ad hoc networks using Cell-DEVS and show how we can model factors such as network coverage and the lowest hop count to wireless gateways for mobile nodes. In our model, mobile nodes move with a constant speed in diagonal-directions. For the purpose of modeling, we have implemented a collision avoidance technique in which the mobile nodes reverse their direction if there is a collision risk with another mobile, static or gateway node. A basic routing behavior is implemented such that every

mobile and static node can find out, if they can reach the gateway and if they can, how many hops away the gateway is. The nodes always choose the neighbor which is closest to the gateway as their next hop. A *coverage* model is also implemented. Every cell (whether or not occupied by an ad hoc node) with an occupied neighbor that can reach a gateway, also has a potential to reach the gateway. Every cell, which has a potential to reach the gateway is considered to be within the coverage area. As the number of hops to the gateway increases so does the total power consumption. Therefore, it is reasonable to put an upper limit on the total number of hops allowed to reach the gateway. In our study we allowed a maximum hop count of five hops. Mobile and static nodes that are 6 or more hops away from the gateway are not allowed to reach the gateway and are therefore not considered within the coverage area. For the experimental evaluation of modeling the behavior of mobile nodes in Cell-DEVS, we have used a simple model in which we fix the size of the neighborhood of a node. We believe that such a neighborhood should be dynamically determined depending on the strength of the signal available at the current location of the node and other factors such as battery power available.

5.1. Mobility model

Mobility behavior of the mobile nodes is implemented by allowing mobile nodes to move in diagonal directions and bounce back whenever they reach the edges of the plane. For example, the top edge cells can only be occupied by mobile nodes with north-east (NE) or north-west (NW) directions. If a mobile node with an NE direction reaches the top edge cell, that mobile node will bounce back from the edge and leave that cell with a SE direction. Similarly, a mobile node that reaches the bottom right corner (with a SE direction) will be bounced back from the corner cell with an NW direction.

Collision avoidance is implemented by checking the state of the cell which is in the direction of the movement of a mobile node. If that cell is not empty, the direction of the mobile node is reversed. If it is empty, then the model checks if there is any other mobile node approaching the same cell from another direction. If there is any, then the direction of the mobile node is reversed. Otherwise, the mobile node is allowed to occupy the cell. There are both static and mobile nodes in our model, and one or more gateways. There is no limit on the number of nodes or gateways in our model. Depending on the initial location and direction of movement of the nodes, different scenarios can be created.

A Cell-DEVS model named *mobilenode* is defined. The coupled model has 20×20 cells; and the surrounding 25 cells forms the neighborhood for each node. There are nine atomic models for inner, top edge, bottom

```
[mobilenode]
type : cell      width: 20        length : 20      height : 3
delay : transportborder : nowrapped

neighbors : (-2,-2,0) (-2,-1,0) (-2,0,0) (-2,1,0) (-2,2,0) (-1,-2,0) (-1,-1,0) (-1,0,0)(-1,1,0)(-1,2,0) (0,-2,0) (0,-
1,0) (0,0,0) (0,1,0) (0,2,0) (1,-2,0) (1,-1,0) (1,0,0) (1,1,0) (1,2,0) (2,-2,0) (2,-1,0) (2,0,0) (2,1,0) (2,2,0) (-2,-
2,-1) (-2,-1,-1) (-2,0,-1) (-2,1,-1) (-2,2,-1) (-1,-2,-1) (-1,-1,-1) (-1,0,-1) (-1,1,-1) (-1,2,-1) (0,-2,-1) (0,-1,-1)
(0,0,-1) (0,1,-1) (0,2,-1) (1,-2,-1) (1,-1,-1) (1,0,-1) (1,1,-1) (1,2,-1) (2,-2,-1) (2,-1,-1) (2,0,-1) (2,1,-1) (2,2,-
1)

zone : cornerUL-rule { (0,0) }
zone : cornerUR-rule { (0,19) }
zone : cornerDL-rule { (19,0) }
zone : cornerDR-rule { (19,19) }
zone : top-rule { (0,1)..(0,18) }
zone : bottom-rule { (19,1)..(19,18) }
zone : left-rule { (1,0)..(18,0) }
zone : right-rule { (1,19)..(18,19) }

[mobility_CA-rule]
rule : 1 1000 {(0,0)=4 and ((-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2)}
rule : 4 1000 {(0,0)=1 and ((1,1) != 0 or (0,2)=3 or (2,2)=4 or (2,0)=2 )}
rule : 3 1000 {(0,0)=2 and ((-1,1) != 0 or (-2,0)=1 or (-2,2)=3 or (0,2)=4)}
rule : 2 1000 {(0,0)=3 and ((1,-1) != 0 or (2,-2)=2 or (2,0)=4 or (0,-2)=1)}
rule : 1 1000 {(0,0)=0 and (-1,-1)=1 and (1,-1)!= 2 and (-1,1)!= 3 and (1,1)!= 4 }
rule : 2 1000 {(0,0)=0 and (1,-1)=2 and (-1,1)!= 3 and (1,1)!= 4 and (-1,-1)!=1 }
rule : 3 1000 {(0,0)=0 and (-1,1)=3 and (1,1)!= 4 and (-1,-1)!= 1 and (1,-1)!=2 }
rule : 4 1000 {(0,0)=0 and (1,1)=4 and (-1,-1)!=1 and (-1,1)!=3 and (1,-1)!=2 }
rule : 5 1000 { (0,0)=5 }
rule : 6 1000 { (0,0)=6 }
rule : 0 1000 { t }
```

Fig. 25. Implementing mobility model in CD++.

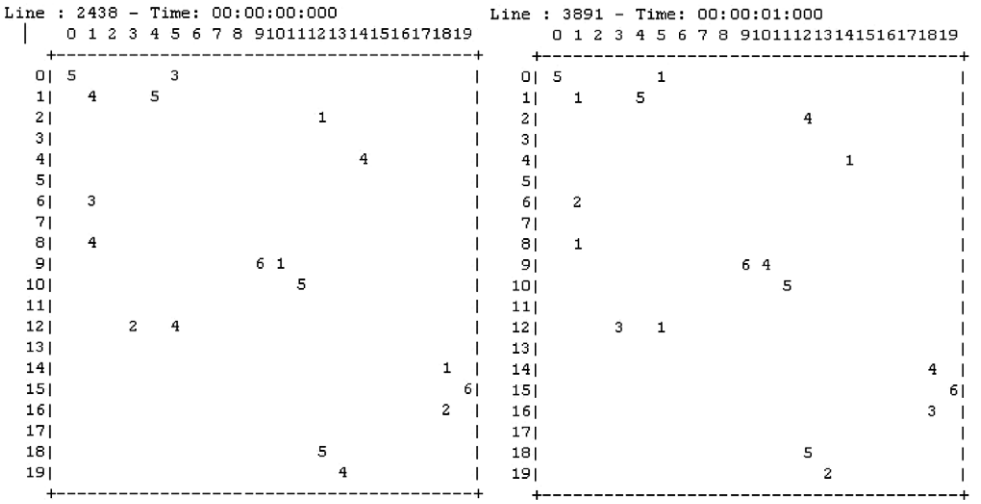

Fig. 26. Nine different collision avoidance scenarios.

edge, left edge, right edge, top left corner, bottom left corner, top right corner and bottom right corner cells. Mobility model for the inner atomic cell is as follows:

$$CD = \langle X, Y, S, E, d, \delta_{\text{int}}, \delta_{\text{ext}}, \tau, \lambda, D \rangle$$

$X = Y = S = \{0$ (Empty), 1 (SE), 2 (NE), 3 (SW), 4 (NW), 5 (Static), 6 (Base Station)$\}$;
$d =$ transport delay, 1000 m; τ showed in the following figure;
δ_{int}, δ_{ext}, λ, D are according to Cell-DEVS definitions.

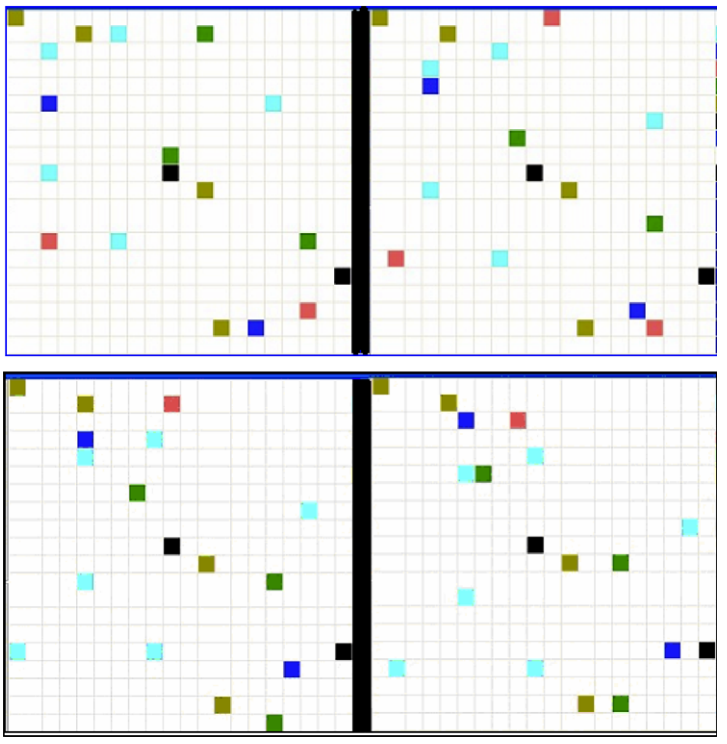

Fig. 27. Mobility, at 02:000, 03:000, 04:000 and 05:000.

The model is implemented in CD++, as presented in Fig. 25. The *mobilenode* model implements mobility, routing and coverage; however, in Fig. 25 we only show mobility related rules. The figure shows rules only for the inner cells. Rule for other cells are not given here due to space considerations (details can be found in [33]).

Here, we use numerical values to represent the model's state variables as follows: $S = 0$ (Empty Cell), 1 (Mobile Node moving in SE direction), 2 (Mobile Node moving in NE direction), 3 (Mobile Node moving in SW direction), 4 (Mobile Node moving in NW direction), 5 (Static Node), 6 (Gateway). As can be seen from the rules in Fig. 25, the value of the cells having static nodes and gateways do not change (as they do not move). The first four rules represent the cases where the current cell is occupied by a mobile node but that mobile node cannot move because either the next cell is already occupied or the next cell is empty but there are other nodes approaching that empty next cell. In such cases the mobile nodes reverse their directions to prevent a collision.

Fig. 26 shows an example of a topology with 2 gateways (value 6), 4 static nodes (value 5), 5 mobile nodes with NW direction (value 4), 2 mobile nodes with SW direction (value 3), 2 mobile nodes with NE direction (value 2) and 3 mobile nodes with SE direction (value 1). Nine different collision scenarios are created from this initial topology. Four of these collisions are between static and mobile nodes (potential collisions between node pairs located at cells: $(0,0)$–$(1,1)$, $(1,4)$–$(0,5)$, $(9,10)$–$(10,11)$, $(18,12)$–$(19,13)$), three of them are between two mobile nodes (potential collisions between node pairs located at cells: $(2,12)$–$(4,14)$, $(6,1)$–$(8,1)$, $(12,3)$–$(12,5)$) and two of them are between a mobile node and a gateway (potential collisions between node pairs located at cells $(14,18)$–$(15,19)$, $(16,18)$–$(15,19)$). Fig. 27 shows that all mobile nodes change their directions at the next time unit in order to avoid collision. For example, at time 03:000, a mobile node approaching the top edge with NW direction, changes its direction to SW. Similarly, in Fig. 27 there is a mobile node which approaches bottom edge cell with SW direction and another which approaches the left edge cell with SW direction. At time 04:000, these two nodes change their directions (after they have reached the edge cells) to NW and SE respectively.

5.2. Hop-count model

We incorporated a hop-count sub-model in the model of Section 5.1. In a hop-count sub-model every mobile and static node determines its neighbor with the least number of hops to the gateway. Minimizing the number of hops decreases the overall transmission power. At the same time, it reduces the overall delay.

```
[hop_cout-rule]
rule : 10 1000 { (0,0)=5 and statecount(6) > 0 }
rule : 20 1000 { (0,0)=5 and ( ( (0,0)=10 and statecount(10) > 0 ) or statecount(10) > 1 ) }
rule : 30 1000 { (0,0)=5 and ( ( (0,0)=20 and statecount(20) > 0 ) or statecount(20) > 1 ) }
rule : 40 1000 { (0,0)=5 and ( ( (0,0)=30 and statecount(30) > 0 ) or statecount(30) > 1 ) }
rule : 50 1000 { (0,0)=5 and ( ( (0,0)=40 and statecount(40) > 0 ) or statecount(40) > 1 ) }
rule : 60 1000 { (0,0)=5 and t }

rule : 0 1000 { (0,0)=0 and (1,1) != 4 and (-1,1) != 3 and (1,-1) != 2 and (-1,-1) != 1 }
rule : 0 1000 { (0,0)=1 and (1,1)=0 and (0,2) != 3 and (2,2) != 4 and (2,0) != 2 }
rule : 0 1000 { (0,0)=2 and (-1,1)=0 and (-2,0) != 1 and (-2,2) != 3 and (0,2) != 4 }
rule : 0 1000 { (0,0)=3 and (1,-1)=0 and (2,-2) != 2 and (2,0) != 4 and (0,-2) != 1 }
rule : 0 1000 { (0,0)=4 and (-1,-1)=0 and (-2,-2) != 1 and (-2,0) != 3 and (0,-2) != 2 }
rule : 6 1000 { (0,0)=6 }

rule : 10 1000 { (0,0)=0 and ( (1,1)=4 and (-1,-1)!=1 and (-1,1)!=3 and (1,-1)!=2 ) and statecount(6) > 0 }
rule : 20 1000 { (0,0)=0 and ( (1,1)=4 and (-1,-1)!=1 and (-1,1)!=3 and (1,-1)!=2 ) and ( ( (1,1,0)!=10 and state-
count(10) > 0 ) or statecount(10) > 1 ) }

...

rule : 10 1000 { (0,0)=4 and ( (-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2 ) and statecount(6) > 0 }
rule : 20 1000 { (0,0)=4 and ( (-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2 ) and ( ( (0,0,0)!=10 and statecount(10)
> 0 ) or statecount(10) > 1 ) }
rule : 30 1000 { (0,0)=4 and ( (-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2 ) and ( ( (0,0,0)!=20 and statecount(20)
> 0 ) or statecount(20) > 1 ) }
rule : 40 1000 { (0,0)=4 and ( (-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2 ) and ( ( (0,0,0)!=30 and statecount(30)
> 0 ) or statecount(30) > 1 ) }
rule : 50 1000 { (0,0)=4 and ( (-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2 ) and ( ( (0,0,0)!=40 and statecount(40)
> 0 ) or statecount(40) > 1 ) }
rule : 60 1000 { (0,0)=4 and ( (-1,-1)!=0 or (-2,-2)=1 or (-2,0)=3 or (0,-2)=2 ) }
```

Fig. 28. Implementing hop count model in CD++.

The model is especially useful for networks where most of the traffic is routed through the wireless gateways.

The model was implemented in CD++. Hop-count model is implemented in a separate plane with values of different cells defined as follows: 10 = (Gateway one hop away), 20 = (Gateway two hops away), 30 = (Gateway three hops away), 40 = (Gateway four hops away), 50 = (Gateway five hops away), 60 = (Gateway more than 5 hops away or in other words unreachable). The hop-count plane rules are shown in Fig. 28. The initial values of all cells are zero. Just like in Section 5.1, our coupled model has 20×20 cell structure. 25 cells surrounding each cell on the hop-count plane plus 25 cells located just below these cells on the mobility plane forms the neighborhood.

The hop-count model works according to the following rules:

- If the current cell value is 0, and either there are no nodes moving to this cell in the mobility plane or there is more than one node trying to move to this cell, the value remains unchanged.
- If a cell is occupied by a gateway in the mobility plane then the cell value does not change.

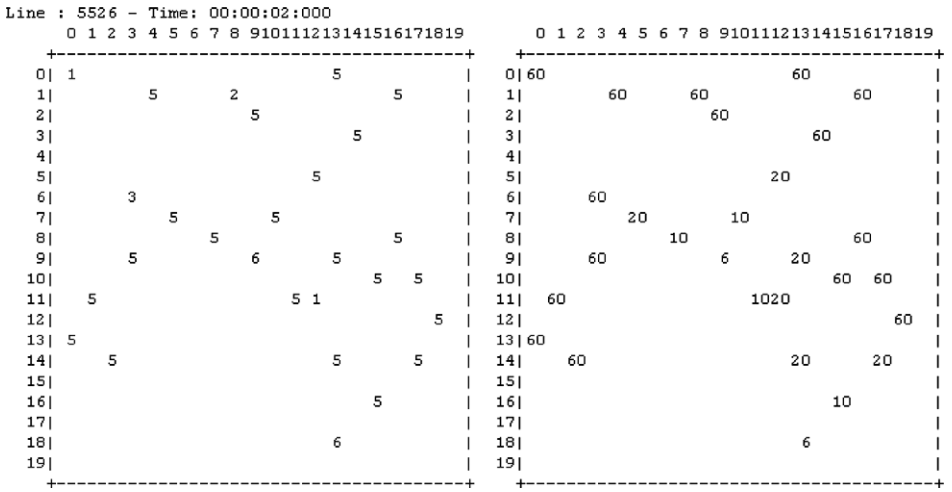

Fig. 29. Mobility (left) and hop-count (right) planes at time 02:000.

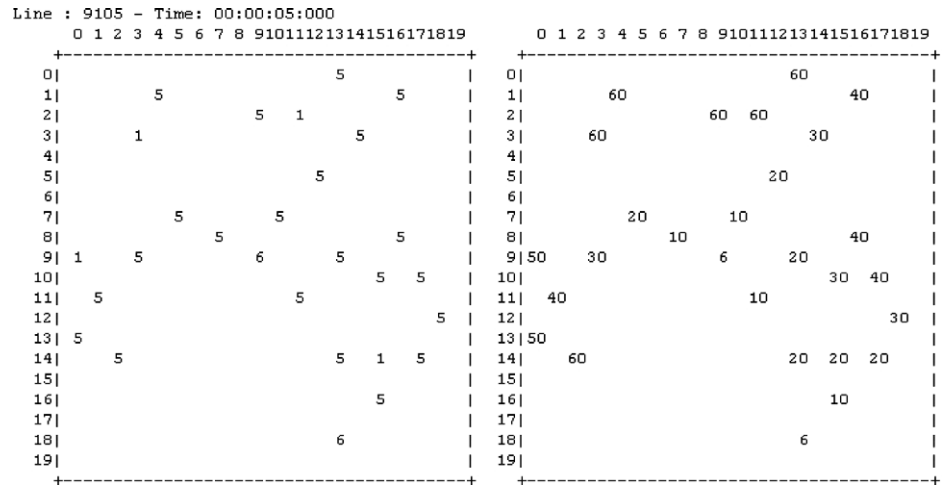

Fig. 30. Mobility (left) and hop-count (right) planes at time 05:000.

- If a cell is occupied by a static node in the mobility plane, then the model checks if there is a gateway in the neighborhood. If there is any then the value of this cell is set to 10 showing that the gateway is one hop away. Otherwise, the model checks if there are any other nodes in the neighborhood. If there are, the model finds the one with the lowest hop-count to the gateway, increments it by ten and sets this value as the current cell value. Otherwise, it is set to 60.
- If a cell is occupied by a mobile node in the mobility plane and if the next cell in the direction of the movement of the mobile node is empty, and there are no other mobile nodes trying to move to the next cell in the mobility plane, then the model sets the cell value to zero.
- If a cell is occupied by a mobile node in the mobility plane and if the next cell in the direction of the movement of the mobile node is not empty, or there are other mobile nodes trying to move to the next cell in the mobility plane, then the model checks if there is a gateway in the neighborhood. If there is any then the cell value is set to 10. Otherwise, the model checks if there are other nodes in the neighborhood. If there are, the model finds the one with lowest hop count value, increments it by ten and sets this value as the current cell value. Otherwise, it is set to 60.

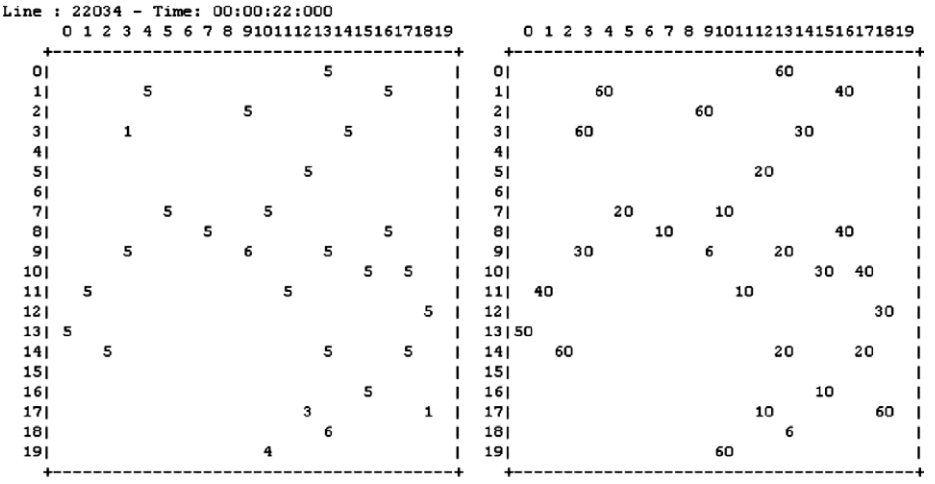

Fig. 31. Mobility (left) and hop-count (right) at time 22:000.

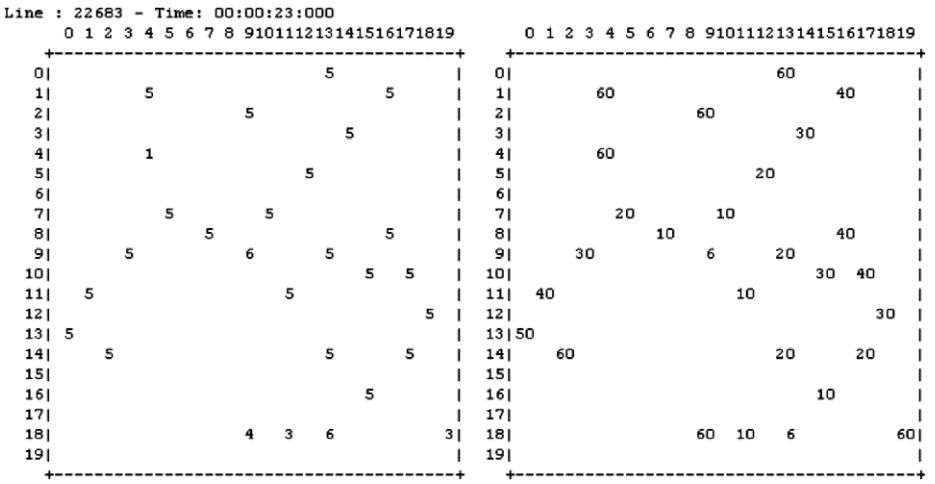

Fig. 32. Mobility (left) and hop-count (right) at time 23:000.

- If a cell is empty and there is only one node moving to this cell in the mobility plane, then the model checks if the gateway is in the neighborhood. If it is, then the cell value is set to 10. Otherwise, the model finds a node in the neighborhood with the lowest hop-count to the gateway, increments it by 10 and sets this value as the current cell value. If no such node is found, it is set to 60.

We created a topology for testing the hop-count plane. The topology of nodes is shown in Figs. 29 and 30. It consists of 2 gateways, 22 static nodes and 4 mobile nodes each with different initial directions. We purposely created loops between nodes and gateways in the initial topology to test if the nodes correctly choose the path with the least number of hops to the gateway.

When the model is executed, as a first step, all nodes lying in the neighborhood of gateways set their hop count value to 10 at Time 01:000. In the second iteration all nodes lying in the neighborhood of cells with hop-count value of 10 set their value to 20, as shown in Fig. 29. This process continues and finally, in the fifth iteration all nodes which are 5 hops away from the gateway set their values to 50. This is shown in Fig. 30. From this point onwards, the hop count values of the cells does not change frequently as after this point the cell values change only due to relative change in position of the mobile nodes. When the mobile nodes change their positions in accordance with the mobility model, the hop count values of the cells are updated. Figs. 29 and 30 also shows that the node located at the right most end of the hop-count plane (i.e. located at cell (12, 18)), can reach both gateways. This node chooses the path with the lowest hop count and sets its value accordingly (i.e., 30 instead of 50).

Fig. 31 shows the state of the model after 22 iterations at Time 22:000. In the figure, when the mobile node located at cell (17, 12) with direction SW and hop count value 10 comes closer to another mobile node located at cell (19, 10), the mobile node located at cell (19, 10) should update its hop-count value to 20. Figs. 31 and 32 show that this does not happen.

The reason is that in the above scenario, when the mobile node located at cell (19, 10) in Fig. 31 with hop-count value of 60 moves to a new cell (18, 9) in Fig. 30, the new cell is not aware of the presence of the node in its neighborhood with a hop-count value of 10. So, the cell does not update its hop-count value accordingly. In real world scenarios as well, when the mobile nodes will move in and out of each other's neighborhood, same transients are expected. However, if mobile nodes communicate with other nodes in their neighborhood or

```
[coverage-rule]
rule : 7 1000 { statecount(6) > 0 }
rule : 7 1000 { statecount(10) > 0 }
rule : 7 1000 { statecount(20) > 0 }
rule : 7 1000 { statecount(30) > 0 }
rule : 7 1000 { statecount(40) > 0 }
rule : 0 1000 { statecount(40) = 0 and statecount(30) = 0 and statecount(20) = 0 and statecount(10) = 0
                and statecount(6) = 0 }
```

Fig. 33. Implementing mobility model in CD++.

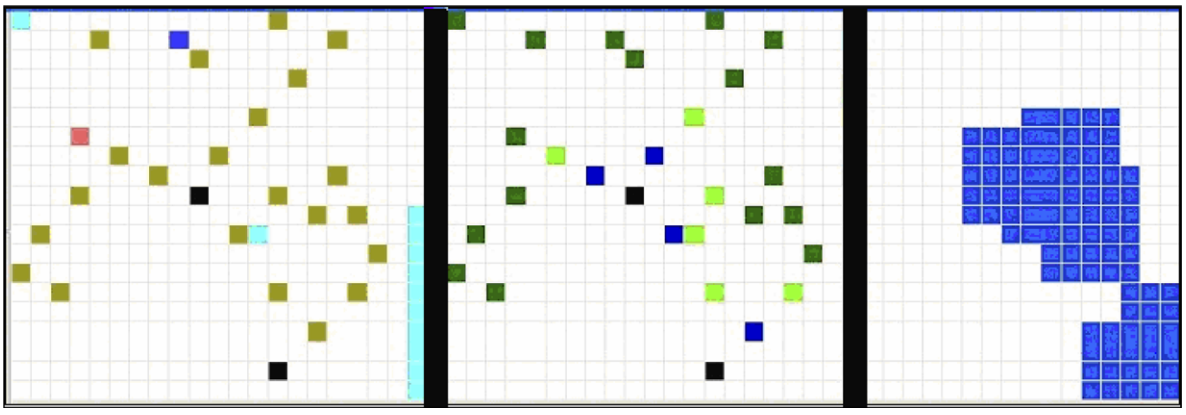

Fig. 34. Mobility (left), hop count (middle) and coverage (right) at time 02:000.

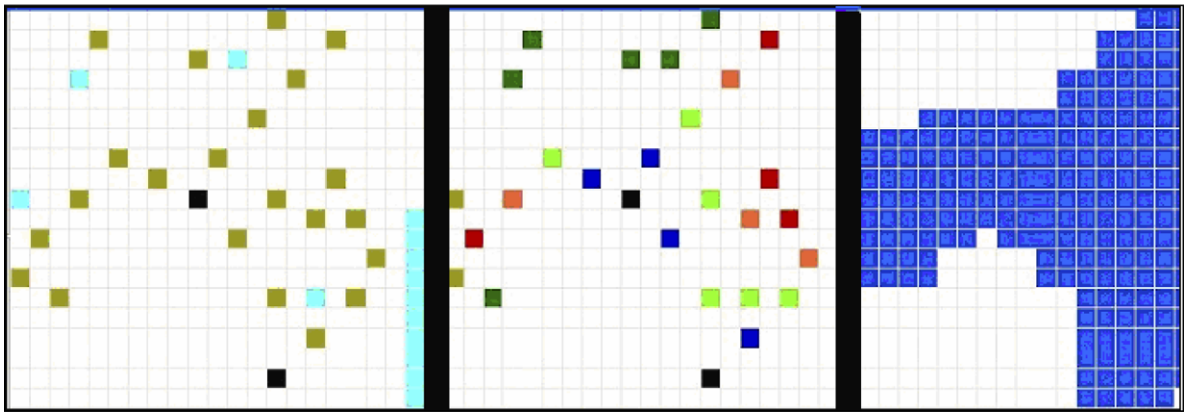

Fig. 35. Mobility (left), hop count (middle) and coverage (right) at time 05:000.

with the gateways before they move to the next cell transients will be much shorter. This scenario is accommodated in our Cell-DEVS model by making execution time for mobility rules higher than the execution time for hop-count state rules. Increasing the neighborhood size can also minimize the transients because with a larger neighborhood, once a mobile node enters into another node's neighborhood, it is expected to stay there long enough to update its connectivity information i.e., its hop-count value. So, by increasing the neighborhood size and/or by executing hop count rules faster than mobility rules, our model can easily prevent transients.

5.3. Coverage model

We implemented a coverage model as a sub-model of our models in the previous sections. The coverage model finds all cells (irrespective of whether or not they are currently occupied by a node) from which a wireless gateway is reachable. All cells that are occupied by the nodes that can reach the gateway and all cells in the neighborhood of these nodes are in the coverage area. Coverage models can help network engineers find the suitable locations for the installation of wireless gateways. It is to be noted that due to the mobility of the nodes the coverage area changes dynamically.

The coverage model was implemented in CD++ on a separate plane. The rules for the coverage plane are shown in Fig. 33 in which a cell value of 7 represents that the cell is in the coverage area while the cell value of 0 represents that the cell is not in the coverage area. Thus in Fig. 33, if there is a gateway or a node with value 10, 20, 30 or 40 within the neighborhood of a cell than value of that cell is changed to 7. Otherwise it remains 0.

Figs. 34 and 35 are the graphical representations of Figs. 31 and 32 respectively. They also include the coverage plane. As can be seen from these figures, when the hop-count plane updates the hop-count values of the nodes, coverage plane updates itself accordingly. Static nodes determine the main shape of the coverage plane. As mobile nodes move around, the shape of the coverage plane continues to change dynamically. This dynamic behavior becomes more pronounced when we increase the number of mobile nodes and decrease the number of static nodes in our model. Another observation from these figures is that there are certain areas in this scenario that irrespective of the movement of the mobile nodes remain out of coverage area. These areas are suitable locations for the installation of gateways so as to ensure coverage.

6. Discussion

Section 3 presented modeling of TCP/IP in DEVS while in Section 4 routing protocols for ad hoc networks are modeled with Cell-DEVS. Section 5 discusses how various characteristics of ad hoc networks such as node

mobility and node coverage model can be represented in Cell-DEVS. Discussion in those sections show how can one easily develop models for quite complex scenarios with a simple schema and how easily those models can be used and tested with several tools available. The models presented in those sections were implemented with just a few lines of rules as opposed to most of the other tools that not only require much more coding but also consultation of various library functions.

Although various models in Sections 3–5 were developed independent of each other, they can be easily combined to test various complex scenarios requiring detailed models of various components and physical characteristics. For example, in Fig. 15 we have shown how DEVS models for TCP/IP can be seamlessly combined with AODV models developed independently in Cell-DEVS. Similarly, one can combine mobility and coverage models in Section 5 to create a more detailed scenario for testing various characteristics of ad hoc networks. Since each model can be developed independently of the other, the process is scalable and allows for selective integration of components as per demands of the testing being conducted. For example, given various models for physical layers, each developed independently, one can plug-in different models for different tests to test the network protocol under varying physical layer characteristics. Independent development of various models also ensures less overhead in testing by allowing unit testing of each model developed. Results in Sections 4 and 5 also shows that Cell-DEVS modeling of ad hoc networks also allowed for easy validation of the results obtained. With the help of CD++ Modeler and VRML GUI one control and visualize each and every step in the simulation. This not only allows for easy debugging but it also facilitates in understanding dynamics of the systems which can help in improvement of various protocols being tested. This feature of modeling in Cell-DEVS makes it one of the most promising techniques among those available for modeling ad hoc networks.

One of the important issues for simulating ad hoc networks is to accurately model node mobility. Various models for node mobility in ad hoc networks have been presented and some of the most widely used relies on the modeling of section of a city where the ad hoc network exists (such as [31]). Although tools such as NS-2 [3] and Op-Net [4] can provide for the implementation of various mobility models, they do not allow the integration of these models with maps of actual cities. In DEVS/Cell-DEVS on the other hand, maps of actual cities and models of actual traffic behavior have been developed [32] and they can be easily integrated with the network simulation models presented here. Such a scheme would provide for simulation and performance evaluations of various protocols for the actual deployment places and hence would make results more valid than other tools. Likewise, in [34] we introduced advanced models of environmental catastrophes (forest fires, pollution, etc.), and in [35,36], we presented different models on emergency evacuation and battlefield scenarios. In all of these cases, wireless communication facilities can improve the work of the first responders. Therefore, having the ability to integrate models of the actual system and the communication facilities that the experts use can result in improved definitions for the routing algorithms and the equipment created for the emergency planners. Integration of such models is straightforward (as it has been shown in [8,16]). By including each sub-model on a different layer of a multidimensional model, the sub-components can interact without further complications. We are currently working on such an integration of our models.

For the experimental evaluation of modeling the behavior of mobile nodes in Cell-DEVS, we have used a simple model in which we fix the size of the neighborhood of a node. We believe that such a neighborhood should be dynamically determined depending on the strength of the signal available at the current location of the node and other factors such as battery power available. However, this limitation in our model is not inherent in Cell-DEVS modeling and one can easily couple models developed in Section 5 with more detailed models developed in Cell-DEVS where the signal strength can be made a function of time as well as location. We are currently working on more detailed models that can incorporate effects of such factors.

We have already discussed how different models developed within DEVS/Cell-DEVS can be easily integrated. Recently there has been some work for integrating tools such as NS-2 and DEVS simulators [37,38] with DEVS. In [37] the authors show how to implement heterogeneous simulators based on the DEVS Bus concept, showing how to integrate DEVS models, NS-2 models and a TCP simulation using the HLA as middleware. In [38], the author shows how to integrate DEVS and NS-2 using the DEVS/C++ simulator. We believe such integration would allow DEVS/Cell-DEVS models to make use of already built libraries in such simulators and would increase their applicability while reducing their development costs.

7. Conclusions

We presented modeling mobility and routing in wireless ad hoc networks using Cell-DEVS. Our research shows that routing protocols can be successfully mapped onto Cell-DEVS and as an example we mapped Ad Hoc On-Demand Distance Vector (AODV) protocol onto Cell-DEVS. Such a modeling can not only provide insights into the dynamics of the system and its reaction to different input stimuli but can also give an in depth analysis of the protocol for different testing conditions such as the number and relative location of nodes, connectivity conditions etc. Moreover, this mapping of the algorithm onto Cell-DEVS resulted in the extension of the algorithm in three different directions (inter-network routing, multicast AODV, routing among multiple pairs of senders and receivers) covered comprehensively in the paper. This shows that, generally speaking, mapping of traditional algorithms onto Cell-DEVS can lead into new ideas in theory and implementation of algorithms.

We also modeled the mobility behavior of the nodes in an ad hoc network consisting of mobile, static and gateway nodes. Network coverage behavior is also successfully implemented, that can provide useful information to network engineers in deciding where to put extra gateways.

As a future work, we plan to incorporate more features into the models. For example, each cell may be assigned an index showing the wireless transmission media changes due to shadowing or fading effects. There will be more reflections and consequently attenuation in the wireless signal in urban areas. It is also possible to insert physical objects, such as mountains, high-rise buildings etc. in the models. Altitude is another factor that can be assigned to each cell to determine wireless transmission index. Addition of such phenomena is straightforward in Cell-DEVS because different models can be easily coupled.

References

[1] T.S. Rappaport, Wireless Communications: Principles and Practice, Prentice-Hall, 2002.
[2] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, M. Gerla, GloMoSim: a scalable network simulation environment, Technical Report 990027, Department of Computer Science, University of California at Los Angles, USA, May 1999.
[3] Kevin Fall, Kannan Varadhan, The ns Manual (formerly ns Notes and Documentation). Available from: <http://www.isi.edu/nsnam/ns/>.
[4] Opnet Technologies, Inc. , Opnet Simulator. <http://www.opnet.com/>.
[5] Pavlosoglou, M. Leeson, R. Green, Spotting emergence in wireless routing protocols, in: Proceedings of the London Communications Symposium, London, UK, September 2003.
[6] R. Subrata, A.Y. Zomaya, Evolving cellular automata for location management in mobile computing networks, IEEE Transactions on Parallel and Distributed Systems 14 (1) (2003) 13–26.
[7] S. Wolfram, A New Kind of Science, Wolfram Media, Inc., 2002.
[8] G. Wainer, N. Giambiasi, Timed Cell-DEVS: modelling and simulation of cell spaces, in: Discrete Event Modeling & Simulation: Enabling Future Technologies, 2001.
[9] B. Zeigler, T. Kim, H. Praehofer, Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Academic Press, 2000.
[10] G. Wainer, CD++: a toolkit to define discrete-event models, Software, Practice and Experience 32 (3) (2002) 1261–1306.
[11] A. Troccoli, G. Wainer, Implementing parallel Cell-DEVS. in: Proceedings of Annual Simulation Symposium, Orlando, FL, USA, 2003.
[12] G. Wainer, W. Chen, A framework for remote execution and visualization of Cell-DEVS models, Simulation 79 (November) (2003) 626–647.
[13] B. Zeigler, H. Cho, J. Lee, H. Sarjoughian, The DEVS/HLA distributed simulation environment and its support for predictive filtering, DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.
[14] C. Zhang, Integrating existing DEVS simulations with the HLA, M.A.Sc. Thesis, Carleton University, 2004.
[15] Y.W. Cho, X. Hu, B. Zeigler, The RTDEVS/CORBA environment for simulation-based design of distributed real-time systems, Simulation 79 (4) (2003) 197–210.
[16] P. MacSween, G. Wainer, On the construction of complex models using reusable components, in: Proceedings of SISO Spring Interoperability Workshop, Arlington, VA, USA, 2004.
[17] E. Glinsky, G. Wainer, Modeling and simulation of systems with hardware-in-the-loop, in: Proceedings of the Winter Simulation Conference, Washington, DC, 2004.
[18] B. Zeigler, Y. Moon, D. Kim, G. Ball, The DEVS environment for high-performance modeling and simulation, IEEE Computational Science and Engineering 4 (3) (1997).
[19] G. Wainer, N. Giambiasi, Application of the Cell-DEVS formalism for cell spaces modeling and simulation, Simulation 71 (1) (2001).

[20] A. Davidson, G. Wainer, ATLAS: a specification language for traffic modelling and simulation, Simulation Modeling, Practice and Theory 14 (3) (2006) 317–337.
[21] C.Y. Lee, An algorithm for path connections and its applications, in: IRE Transaction on Electronic Computers, September 1961, pp. 345–365.
[22] C. Perkins, E. Belding-Royer, S. Das, Ad Hoc On-Demand Distance Vector (AODV) Routing Protocol, IETF Network Working Group, RFC 3561, July 2003.
[23] C. Hochberger, R. Hoffmann, Solving routing problems with cellular automata, in: Proceedings of the Second Conference on Cellular Automata for Research and Industry, Milan, Italy, 1996.
[24] C.E. Perkins, P. Bhagwat, Highly dynamic DSDV for mobile computers, in: Proceedings of the ACM Conference on Communications Architectures, Protocols and Applications, London, UK, August 1994, pp. 234–244.
[25] D. Johnson, D. Maltz, Y. Hu, The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR), IETF MANET Working Group Internet Draft, April 2003.
[26] J. Raju, J.J. Garcia-Luna-Aceves, A new approach to on-demand loop-free multipath routing, in: Proceedings of the 8th International Conference on Computer Communications and Networks (IC3N), Boston, MA, USA, October 1999, pp. 522–527.
[27] M. Corson, J. Macker, Mobile Ad Hoc Network (MANET) Routing Protocol and Performance Evaluation Considerations, IETF Networking Group RFC 2501, January 1999.
[28] V. Naoumov, T. Gross, Simulation of large ad hoc networks, in: Proceedings of the ACM 6th International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems, San Diego, CA, USA, September 2003, pp. 42–48.
[29] H. Hellbrück, Stefan Fischer, Towards analysis and simulation of ad hoc networks, in: Proceedings of the 2002 International Conference on Wireless Networks (ICWN'02), Las Vegas, NV, USA, June 2002, pp. 69–75.
[30] M. Ahmed, K. Yonis, M. Elshafei, G. Wainer, Building library of network protocols in CD++, in: Proceedings of the 38th IEEE/SCS Annual Simulation Symposium, San Diego, CA, USA, 2005.
[31] RFC-editor Official Internet Protocol Standards, [Online]. <ftp://ftp.rfc-editor.org/in-notes/rfc791.txt> (accessed 24.9.2003).
[32] B. Balya, U. Farooq, G. Wainer, Modeling ad-hoc networks using Cell-DEVS models, in: Proceedings of the 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'04), San Jose, CA, 2004.
[33] U. Farooq, B. Balya, Modeling routing in wireless ad hoc networks using Cell-DEVS, Internal Report, Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada, December 2003.
[34] G. Wainer, Applying Cell-DEVS methodology for modeling the environment, Simulation 82 (10) (2006) 635–660.
[35] R. Madhoun, G. Wainer, Developing defense applications using DEVS/Cell-DEVS, SCS Journal of Defense Modeling and Simulation 2 (3) (2005) 121–143.
[36] E. Poliakov, G. Wainer, J. Hayes, M. Jemtrud, A busy day at the SAT building, in: Proceedings of AIS 2007, Artificial Intelligence, Simulation and Planning, Buenos Aires, Argentina, 2007.
[37] Y.J. Kim, J.H. Kim, T.G. Kim, Heterogeneous Simulation Framework Using DEVS BUS, Simulation 79 (1) (2003) 3–18.
[38] T. Kim, Devs-Ns2 Environment: an integrated tool for efficient networks modeling and simulation, Master's Thesis, University of Arizona, 2006.