

**DEVS-NS2 ENVIRONMENT; AN INTEGRATED TOOL FOR  
EFFICIENT NETWORKS MODELING AND SIMULATION**

by

**Taekyu Kim**

---

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements  
For the Degree of

MASTER OF SCIENCE  
WITH A MAJOR IN COMPUTER ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

**2006**

## STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: \_\_\_\_\_

## APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

\_\_\_\_\_  
Bernard P. Zeigler, Ph.D.  
Professor of Electrical and Computer Engineering

\_\_\_\_\_  
Date

## **ACKNOWLEDGEMENTS**

I would like to express my truthful gratitude to Professor Bernard P. Zeigler, my thesis advisor, for his guidance and support during the course of this research period. I would also like to thanks to my thesis Co-Advisor Dr. Doohwan Kim for his assistance.

Specially, my sincere appreciation goes to Dr. Moon Ho Hwang for his big contribution to this study. Not only he supports his work, Open DEVS C++, to this research, but also he provides me direction and insight on numerous occasions during this work. He is deserved for my respect and thank.

I thank my wife, Jiyoung Kim, for her constant support and appreciation. I am indebted to her and my parents for all the encouragement.

## TABLE OF CONTENTS

<b>LIST OF FIGURES .....</b>	<b>6</b>
<b>LIST OF TABLES .....</b>	<b>8</b>
<b>ABSTRACT .....</b>	<b>9</b>
<b>CHAPTER 1. INTRODUCTION .....</b>	<b>11</b>
1.1. Motivation and Goals.....	11
1.2. Organization of the Thesis .....	12
<b>CHAPTER 2. BACKGROUND .....</b>	<b>14</b>
2.1. Discrete Event Simulation .....	14
2.1.1. Fundamentals of Computer Simulation .....	14
2.1.2. Discrete Event System Specification .....	17
2.2. NS-2 .....	21
2.2.1. NS-2 Architecture .....	22
2.2.2. NS-2 Event Scheduler.....	24
<b>CHAPTER 3. INTEGRATION OF DEVS AND NS-2 .....</b>	<b>26</b>
3.1. Synchronization .....	26
3.1.1. NS-2 Event Queue Agent.....	28
3.2. DEVS Model Behaviors .....	30
3.2.1. Sensor Node Atomic Model.....	32
3.2.2. Moving Object(Tank) Atomic Model .....	36
3.2.3. Collision Checker Atomic Model .....	38
3.3. Scenario for confirming the integration.....	39
3.3.1. Node Configuration .....	39

## **TABLE OF CONTENTS - Continued**

3.3.2. Network Topology .....	46
3.3.3. Simulation Results .....	47
3.4. Reference Comparison.....	58
3.4.1. DEVS BUS framework.....	58
3.4.2 Comparison between DEVS-NS2 and DEVS BUS framework .....	60
3.5. Comparison between DEVS-NS2 and OPNET .....	62
 <b>CHAPTER 4. EXPERIMENT AND RESULTS .....</b>	<b>66</b>
4.1. Scenario.....	66
4.2. DEVS Model Behavior.....	68
4.2.1. The Sensor Node Model .....	70
4.2.2. The Base Node Model .....	72
4.2.3. The Tank Model.....	74
4.2.4. The Missile Model.....	74
4.2.5. The Collision Checker Model.....	76
4.2.6. The Explosion Checker Model .....	78
4.2.7. The NS-2 Event Queue Agent Model.....	80
4.3. Simulation Results .....	80
 <b>CHAPTER 5. CONCLUSION AND FUTURE WORKS.....</b>	<b>88</b>
5.1. Conclusion .....	88
5.2. Future works .....	89
 <b>REFERENCES .....</b>	<b>91</b>

## LIST OF FIGURES

Figure 1. Classification of Computer Simulation .....	16
Figure 2. System representation of atomic model .....	18
Figure 3. An example of coupled model.....	21
Figure 4. the duality of C++ and OTcl.....	23
Figure 5. NS-2 architecture.....	24
Figure 6. NS-2 event scheduler.....	25
Figure 7. The conceptual architecture of DEVS and NS-2.....	27
Figure 8. NS-2 Event Queue Agent Model.....	29
Figure 9. DEVS models in the example of wireless sensor network.....	31
Figure 10. DEVS Sensor Node Model and NS-2 Sensor Node Model .....	33
Figure 11. Sensor Node Model State Diagram .....	34
Figure 12. Moving Object(Tank) Model State Diagram .....	36
Figure 13. Collision Checker Model State Diagram.....	39
Figure 14. Node Configuration .....	41
Figure 15. Schematic of the Sensor Node.....	43
Figure 16. Schematic of the relation with DEVS and NS-2 Sensor Node.....	45
Figure 17. Network Topology and pre-defined tank route .....	46
Figure 18. Simulation Initialization .....	48
Figure 19. Simulation Console when a sensor node detects a tank .....	51
Figure 20. Simulation Console when a tank disappears from the sensing area .....	52
Figure 21. The remaining energy for each node .....	53
Figure 22. The total number of generated packets in the simulation.....	55
Figure 23. The simulation result at the sensor node 0, 2, 4 .....	56
Figure 24. Pre-defined scenario code.....	57
Figure 25. DEVS BUS framework [6].....	59
Figure 26. Schematic of the scenario .....	67
Figure 27. Schematic Architecture of the Modeling.....	69

## LIST OF FIGURES - Continued

Figure 28. Sensor Node Model State Diagram .....	70
Figure 29. Base Node Model State Diagram .....	73
Figure 30. Missile Model State Diagram.....	75
Figure 31. Collision Checking Method.....	77
Figure 32. Explosion Checker Model State Diagram .....	79
Figure 33. Simulation Initialization .....	82
Figure 34. Simulation Console with no alive packet sending.....	83
Figure 35. Simulation Console with 1 second alive packet interval.....	84
Figure 36. Total number of packets .....	85
Figure 37. The energy consumption for each sensor node .....	86
Figure 38. The total energy consumption .....	86

## LIST OF TABLES

Table 1. The simulation results for the sensor nodes to detect the tank .....	50
Table 2. The energy remains during the simulation process .....	54
Table 3. Comparison between DEVS-NS2 and DEVS BUS framework .....	61
Table 4. Comparison of DEVS-NS2 and OPNET .....	65



## **ABSTRACT**

The new DEVS-NS2 modeling and simulation environment supports both high and low levels of abstraction network modeling and simulation. DEVS (Discrete Event System Specification) is a well-defined mathematical formalism specification for structure and behavior of dynamic systems. The NS-2 is a discrete event network simulator, whose primary use is intended to build and run various detailed network models and protocols such as TCP/IP, satellite links, and wireless networks. By combining the two powerful modeling and simulation systems, the significant benefits attained by the interoperable simulation of DEVS and NS-2 are reduction of the cost, increased high and low level modeling power, and enhanced reusability.

To integrate the systems seamlessly, two major challenges are addressed. The first challenge is to synchronize the different ways of handling event schedules by the two simulation systems. We illustrate how the simulation time advances of DEVS and NS-2 are synchronized with each other. The latter problem is related to assigning the appropriate level of model structure and behavior within the combined system. The details of low level network with protocol and component description is modeled by NS-2 while DEVS serves as controller by modeling the high level behavior (e.g. use case scenario builder) of target network models and interaction of the associated actors.

In this thesis, we take an example of wireless sensor networks to describe our approach to the development process for modeling and simulation in DEVS-NS2

environment. This example is extended to demonstrate an effective way to make a decision on the appropriate level of sensor node's behavior. This leads to the discussion of tradeoffs between energy efficiency and effectiveness of decision making for the sensor network.

## CHAPTER 1. INTRODUCTION

### *1.1. Motivation and Goals*

Recently, the huge growth of computer networks and communication systems has spurred the development of Modeling and Simulation (M&S) frameworks for both network protocols and applications. There exists a well known network simulator that is widely used by academia; it is the Network Simulator Version 2 (NS-2) [1]. NS-2 is a discrete event driven network simulator for various networking models. Considering only network behaviors, NS-2 supports packet transmitting related studies such as network protocol development, transmitting delay, and so on. Discrete Event System Specification (DEVS) [2] is a system specification modeling structural architecture which is based on a well-defined mathematical formalism. However, DEVS is weak at computer network simulations in that it doesn't have full ranges of detailed network protocol and component nodes compared to NS-2 or OPNET [3]. Expanding network protocols in DEVS requires high initial development cost so that the interoperable simulation of DEVS and NS-2 is expected to reduce modeling development cost. The combination of DEVS and NS-2 has two advantages in terms of modeling power and reusability, which are the main objectives of this thesis.

These two simulators, DEVS and NS-2, have their own event scheduling methods. Because of this, time synchronization is the most challenging research in integrating them. In this thesis, we synchronize DEVS and NS-2 first. The interoperable simulation of DEVS and NS-2 is named as DEVS-NS2. An example of

wireless sensor networks is modeled and simulated. The behavior of a sensor node's application and its environmental behaviors such as battle fields are defined in DEVS modeling and the roles of networking protocol behaviors are assigned to NS-2 since NS-2 has well-designed network protocol libraries. In other words, DEVS models reside on the top of NS-2 layered network protocol models so that the roles of DEVS models and NS-2 models are distinguished and adjusted. Consequently, modeling power is increased and modeling development cost is reduced. Finally, a feasible sensor node's behavior for effective decision making is introduced.

The machine used in this research is the Microsoft Windows XP operating system with Intel Pentium 4 processor. Because NS-2 simulator version 2.28 is to be integrated with DEVS in this thesis, and is normally run on Unix/Linux systems, we develop the DEVS-NS2 under Cygwin [4] which is a Linux-like environment for Microsoft Windows. In this research, the DEVS formalism is used and we use the Open DEVS C++ [5] which is developed and expanded by Dr. M. Hwang.

## ***1.2. Organization of the Thesis***

The remainder of this thesis is organized as follows. Chapter 2 briefly introduces background knowledge necessary for the remaining chapters. It covers modeling and simulation based on the DEVS paradigm and NS-2.

Chapter 3 describes the time synchronization method in DEVS-NS2 that is the key in integrating DEVS and NS-2. An example and its DEVS models are shown in

order to prove that the DEVS-NS2 is integrated well and is risk free. The last part of chapter 3 compares DEVS-NS2 with the previous study, DEVS BUS framework [6].

Chapter 4 presents an example in wireless sensor networks. Models that are used in the example are described and our proposed sensor node's behavior is evaluated in terms of both energy efficiency and decision making compared to one of the conventional sensor node's.

Chapter 5 summarizes and concludes this thesis. In addition, future works are discussed.

## **CHAPTER 2. BACKGROUND**

In this chapter, Discrete Event System Specification (DEVS) formalism is introduced in the first half. The second half of this chapter introduces Network Simulator Version 2 (NS-2). The overall NS-2 architecture and the event scheduler of NS-2 are explained for the purpose of understanding how to integrate with DEVS and NS-2 in chapter 3.

### ***2.1. Discrete Event Simulation***

#### **2.1.1. Fundamentals of Computer Simulation**

Computer simulation is an activity of representing the temporal behavior of a physical or a conceptual system for a specific period of time. A simulation model is a specification representing the system in terms of a set of states, events, and behavior functions. Simulation time can be slower, faster, or equal to physical time. Also, time resolution can be arbitrarily defined[7].

During simulation, the current status of a model is represented by a state. A state transition occurs just before initiating or after completing a particular behavior. A state feasibility test may be involved before a state transition happens. An event is a data object that is produced and consumed by simulation components: e.g., logical simulator and coordinator. If necessary, a set of events is exchanged among those components in order to complete a simulation task. A behavior function is invoked

when events are received or produced by a model or a specific behavior of the model is to be performed.

Simulation is classified into continuous simulation and discrete simulation according to the state transition occurrence interval. During a simulation, if a state transition occurs continuously in time, the simulation is a continuous simulation. While, if state transitions happens in discrete time, the simulation is called a discrete simulation. In a discrete simulation, if state transitions occur in term of discrete time interval (or time steps), the simulation is referred to as a time driven discrete simulation (or discrete-time driven simulation). An event-driven discrete simulation (or discrete-event driven simulation) is defined if state transitions happen based on event activities. Figure 1 depicts the classification of a computer simulation.

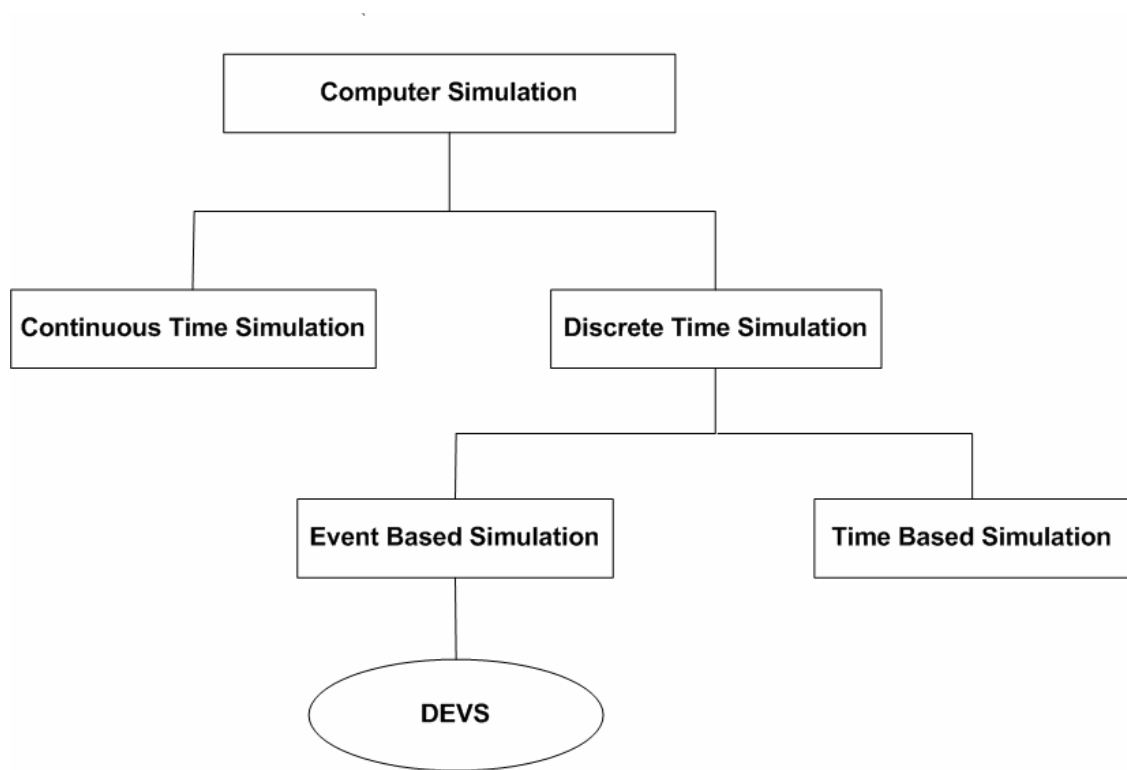


Figure 1. Classification of Computer Simulation

Depending on the simulation time synchronization scheme, a simulation is viewed as a conservative or an optimistic activity at a specific time. All simulation activities are completed before advancing time and time must be synchronized in the conservative scheme[8, 9]. In the optimistic scheme, time does not need to be synchronized globally and simulation activities at a particular time need not all be completed before advancing time. Only when a time causality problem occurs, time needs to be synchronized [10, 11]. Conservative schemes guarantee all activities are performed without any time causality problems. By loosening the time causality constraint, optimistic schemes perform better for certain simulation problems that contain



a high degree of parallelism between simulation models. However, it requires additional memory to keep information in regards to activities that occurred at a previous time. When time causality problems happen, current simulation time rolls back to a previous time that did not violate time causality. Generally, the performance of the simulation is not directly associated with a simulation time synchronization scheme but instead is related to the nature of the given simulation problem[7].

### 2.1.2. Discrete Event System Specification

The Discrete Event System Specification (DEVS) is a formalism providing a mean of specifying a mathematical object called a system. It also allows building modular and hierarchical model compositions based on the closure-under-coupling paradigm. The DEVS modeling approach captures a system's structure from both functional and physical points of view. A system is described by a set of input/output events and internal states along with behavior functions regarding event consumption/production and internal state transitions. Generally, models are considered as either atomic models or coupled models. The Atomic model can be illustrated as a black box having a set of inputs( $X$ ) and a set of outputs( $Y$ ). It describes interface as well as data flow between the atomic model itself and other DEVS models. The Atomic model also specifies a set of internal states( $S$ ) with some operation function (i.e., external transition function ( $\delta_{ext}$ ), internal transition function ( $\delta_{int}$ ), output function ( $\lambda$ ), and time advance function ( $ta()$ )) to describe the dynamic behavior of the model. Figure 2 illustrates the system representation of an atomic

model [12].

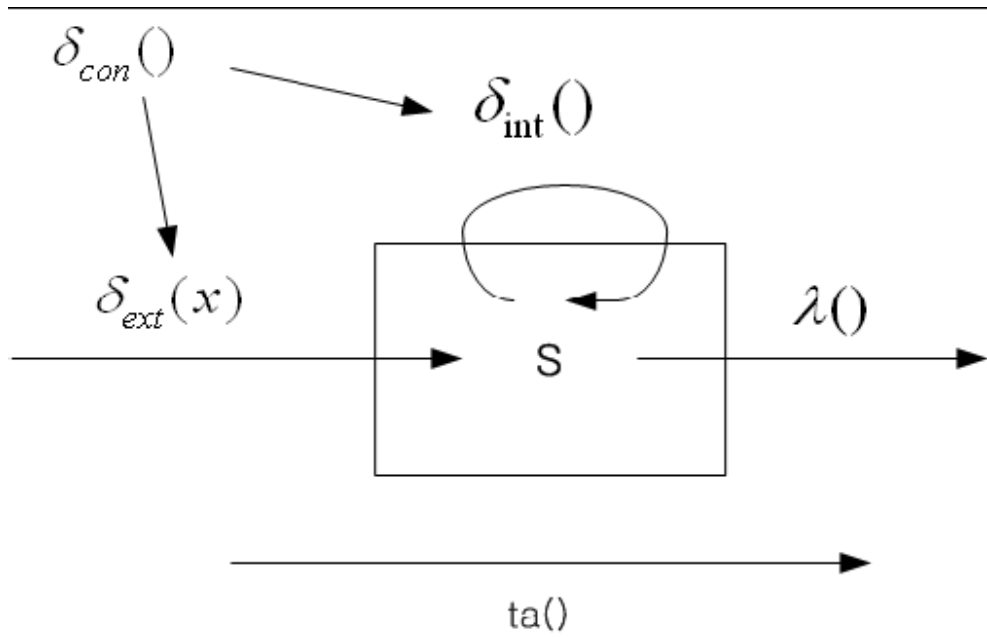


Figure 2. System representation of atomic model

The external transition function ( $\delta_{ext}$ ) carries the input and changes the system states. The internal transition function ( $\delta_{int}$ ) changes internal variables from the previous state to the next when no events have occurred since the last transition. The output function ( $\lambda$ ) generates an output event to outside models in the current state. The time advance ( $ta()$ ) function adjusts simulation time after generating an output event. The Atomic model is specified as follows:

Atomic model:

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

where,

$X$ : set of external input events;

$S$ : set of sequential states;

$Y$ : set of outputs;

$\delta_{\text{int}} : S \rightarrow S$  : internal transition function

$\delta_{\text{ext}} : Q \times X^b \rightarrow S$  : external transition function

Where,  $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ ; is the set of total states

$e$  is the elapsed time since last state transition

$X^b$  is a set of bags over elements in  $X$ ,

$\lambda : S \rightarrow Y^b$  : output function generating external events at the output

$ta : S \rightarrow R_{0,\infty}^+$  : time advance function;

Basic models may be coupled in the DEVS formalism to form a coupled model. A coupled model is the major class which embodies the hierarchical model composition constructs of the DEVS formalism [13]. A coupled model is defined by specifying its component models, called its components, and the coupling relations which establish the desired communication links. A coupled model tells how to couple (connect) several component models together to form a new model. Two major activities involved in coupled models are specifying its component models and defining the couplings which create the desired communication networks. A coupled model is

defined as follows:

Coupled Model:

$$DN = \langle X, Y, D, \{M_{ij}\}, \{I_i\}, \{Z_{i,j}\} \rangle$$

where,

X: set of external input events;

Y: a set of outputs;

D: a set of components names;

for each  $i$  in  $D$ ,

$M_i$  is a component model

$I_i$  is the set of influences for  $i$

for each  $j$  in  $I_i$

$Z_{i,j}$  is the  $i$ -to- $j$  output translation function

A coupled model template contains the following information [14]:

- The set of components
- The set of input ports through which external events are received
- The set of output ports through which external events are sent
- The coupling specification consisting of:
  - ▶ The external input coupling(EIC) connects the input ports of the Coupled model to one or more of the input ports of the components
  - ▶ The external output coupling(EOC) connects the output ports of the

components to one or more of the output ports of the coupled model

- Internal coupling(IC) connects output ports of components to input ports of other components

Figure 3 is an example of coupled model.

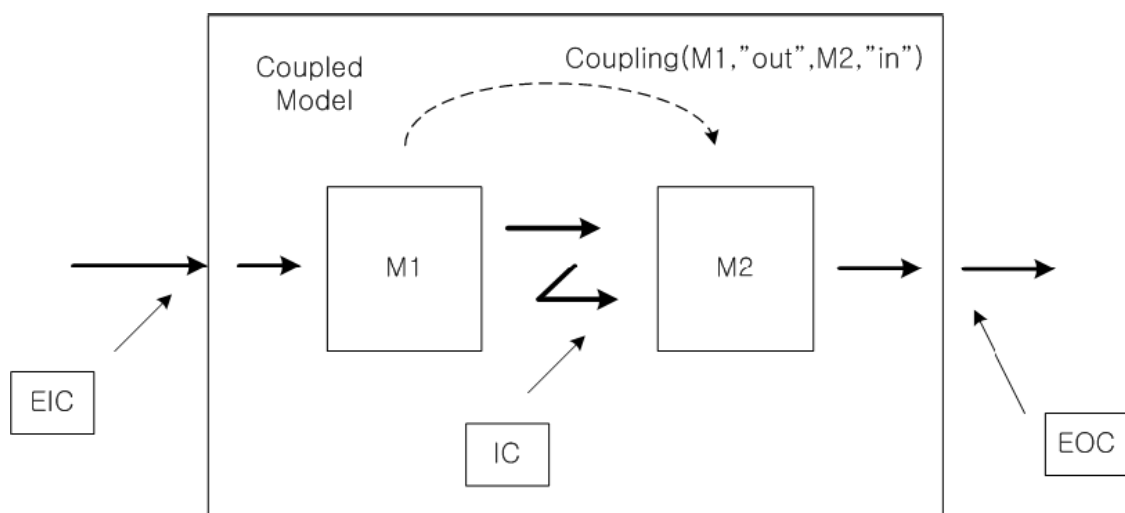


Figure 3. An example of coupled model

## 2.2. NS-2

Network Simulator Version 2 (NS-2) is a discrete event driven simulator. It has been developed at the University of California, Berkeley. It is object-oriented and is designed primarily for local and wide area network simulations. Although it provides a lot of well organized documents, it is not easy to use because it has been extended by many developers so that the architecture of NS-2 is very complicated. In this chapter,

some basic ideas of how the NS-2 simulation engine works, how to initialize simulation setup, and how to analyze simulation results are introduced.

### **2.2.1. NS-2 Architecture**

NS-2 is an object-oriented simulator which is written in C++ and OTcl. The advantages of an object-oriented system are reusability and easy maintenance. While there exists the drawbacks of performance (speed and memory) inefficiency and careful planning of modularity. The reason why NS-2 is written with C++ and OTcl separately is to compromise between modularity and speed. C++ is used for data while OTcl is employed for control. Because C++ is much faster than OTcl, C++ is used for run-time speed critical tasks such as detailed protocols. But, once network components which are written in C++ codes are compiled, no change can be made. OTcl is used for control such as an interpreter since it runs slow but changes quickly. Figure 4 shows the duality of C++ and OTcl.

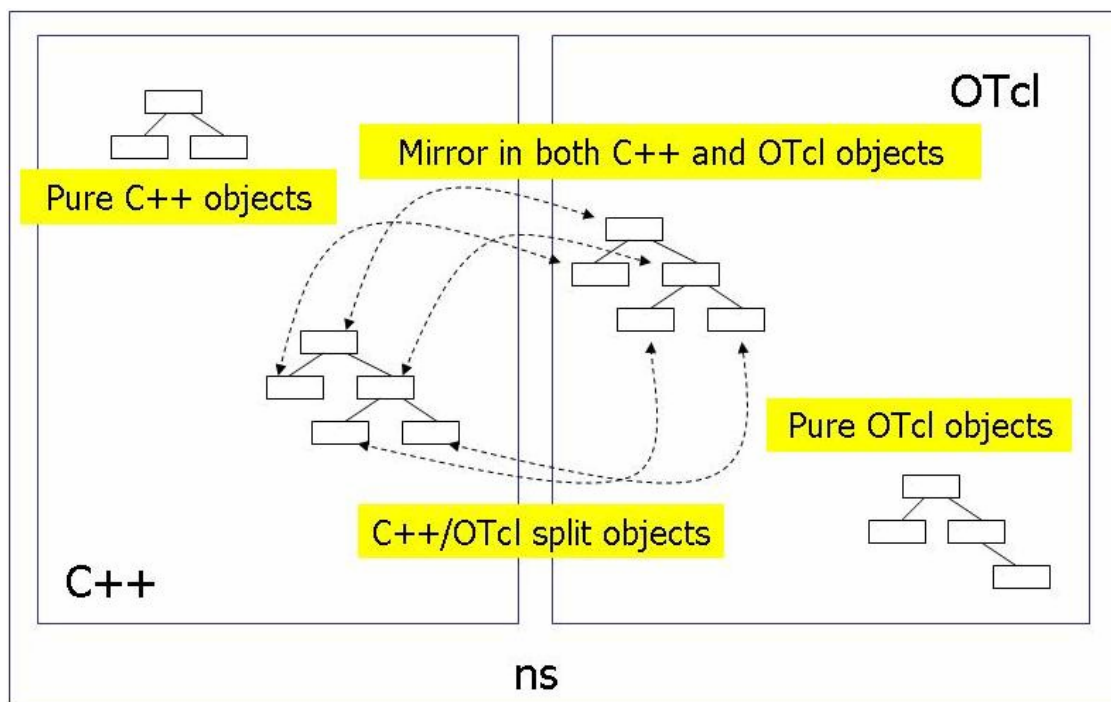


Figure 4. the duality of C++ and OTcl

Figure 5 depicts the architecture of NS-2. In Figure 5, users make simulation scripts using Tcl language in order to design and simulate. NS-2 is composed with C++ objects which are the network components and the event scheduler, OTcl linkage that is implemented in TclCL, and OTcl.

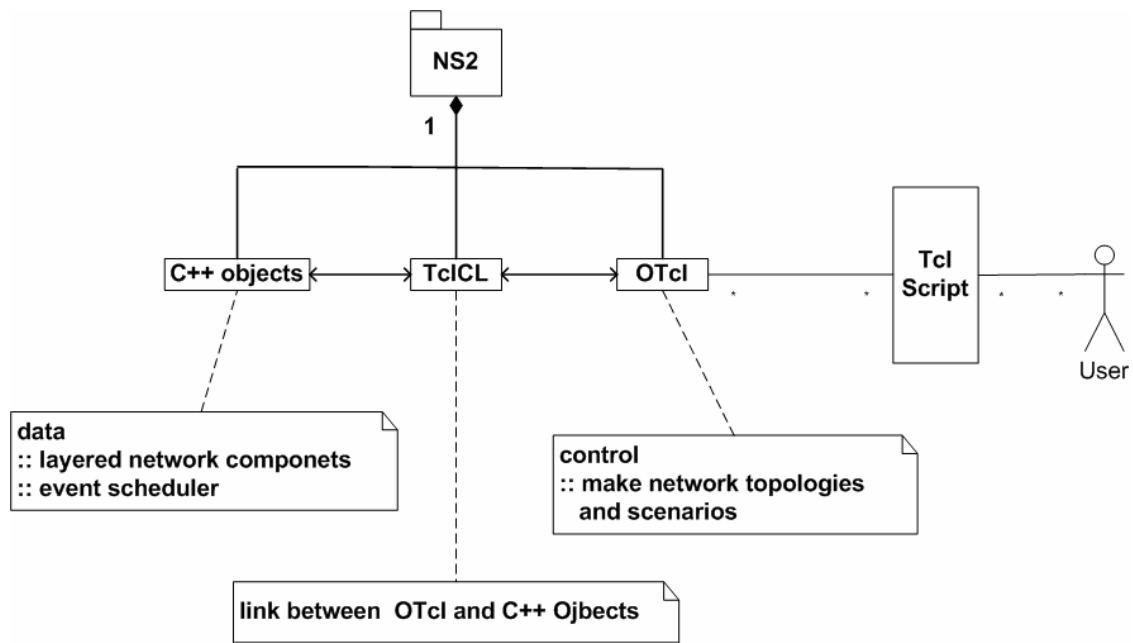


Figure 5. NS-2 architecture

### 2.2.2. NS-2 Event Scheduler

In this section, the NS-2 event scheduler is discussed. The event scheduler is one of the independent C++ objects and it is a discrete event scheduler. Basically, the NS-2 event scheduler is based on a logical time scheduling. Figure 6 depicts a simple overview of the NS-2 event scheduler.



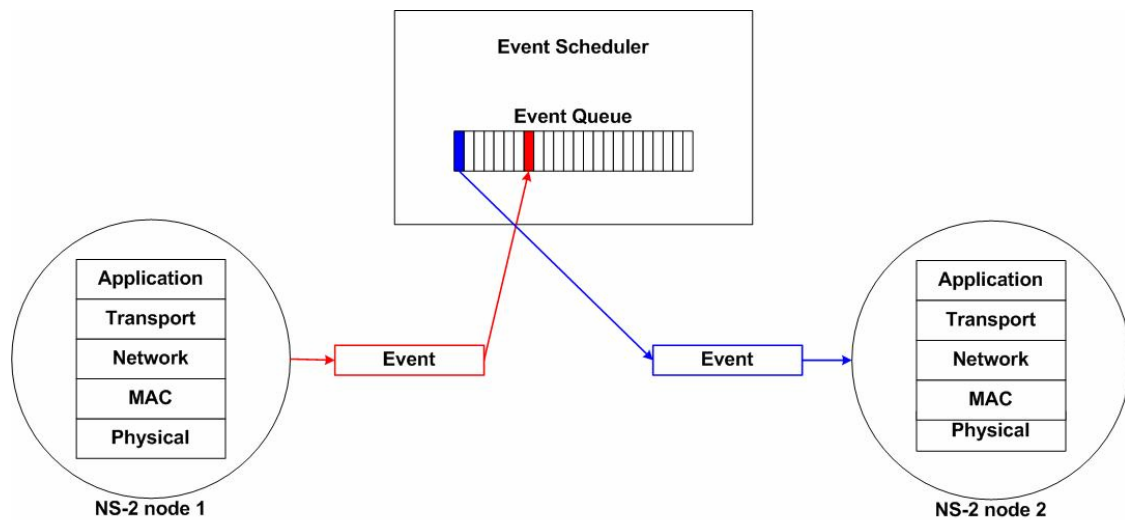


Figure 6. NS-2 event scheduler

The event scheduler has its own event queue and there is only one event queue during a simulation. Every network component such as network nodes has a link to the event scheduler. Once a network component creates an event, the network component puts the event into the event queue. As a simulation goes on, the event scheduler looks at the very first event in the event queue and assigns the first event to appropriate network components at a time which is included in the event. The background knowledge which is presented in this section is helpful for a better understanding of the integration method of the DEVS-NS2.

## **CHAPTER 3. INTEGRATION OF DEVS AND NS-2**

### ***3.1. Synchronization***

The synchronization is the high priority issue to integrate DEVS with NS-2. Both DEVS and NS-2 have their own simulation time management. It is not allowed that either DEVS or NS-2 goes faster than the other. Whenever an event occurs in either DEVS or NS-2, the one which gets an event has to inform the other that an event has happened in it, and the other needs to synchronize with the simulator which caused the event. Initially, we have to make sure that they use same time unit and time mechanism. Because NS-2 uses the logical simulation time and the time unit is a second, we use the logical time and the second for a time unit in DEVS. Figure 7 shows basic conceptual architecture of a heterogeneous simulation between DEVS models and NS-2 models.

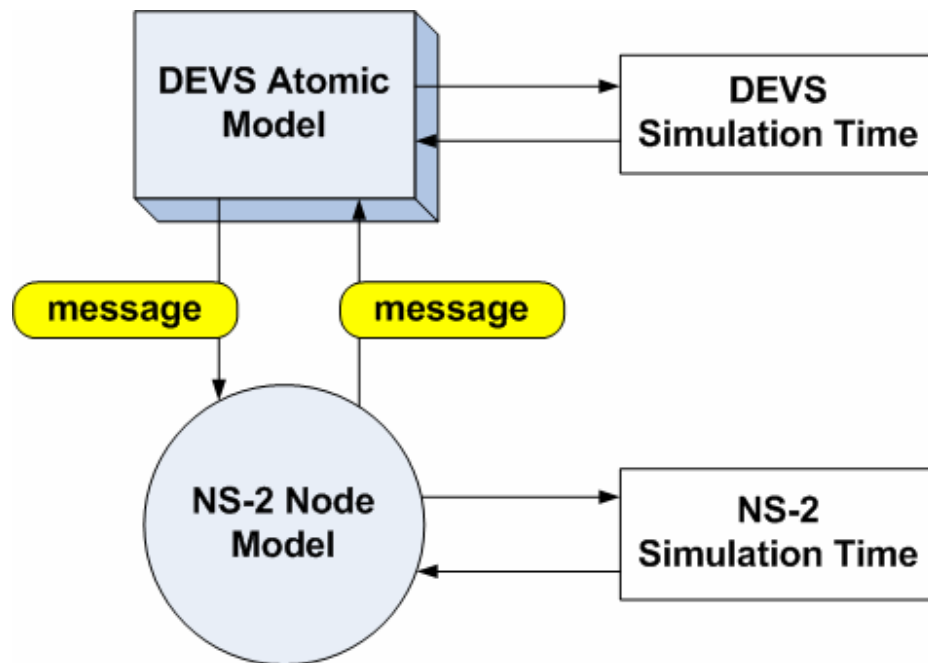


Figure 7. The conceptual architecture of DEVS and NS-2

Figure 7 represents the conceptual architecture of DEVS and NS-2 for the point of view of time synchronization. The DEVS simulation time is not a problem when a DEVS atomic model has an event because the DEVS simulation time and a DEVS atomic model work together in the same coordinator. However, the NS-2 simulation time should be updated as soon as an event occurs in a DEVS atomic model. If a DEVS atomic model that has an event doesn't let an NS-2 node model know that it has an event, there is no way for a NS-2 node model to know that the DEVS simulation time had been advanced. As a result, whenever a DEVS atomic model gets an event, it should send a message regarding an event occurrence to update the NS-2 simulation time. That's the way of time synchronization between a DEVS and an NS-2. Similarly as DEVS atomic models do, NS-2 node models must inform their matching DEVS atomic model that the

simulation time is updated whenever they get events. There exists a DEVS atomic model which deals with the time synchronization between DEVS and NS-2. It is named as “NS-2 Event Queue Agent”. The NS-2 Event Queue Agent model connects to the event queue of NS-2 through a DEVS NS-2 interface. As a result, this model triggers NS-2 to put events into the event queue whenever DEVS models get events. This is discussed in section 3.1.1.

### **3.1.1. NS-2 Event Queue Agent**

This model is the most important one of this thesis. This atomic model connects to the event queue of NS-2. So, this model controls the NS2’s schedule. Once the DEVS coordinator calls this model through the DEVS process, this model dispatches the first event and processes it. Figure 8 shows the relation with the NS-2 Event Queue Agent model, the Sensor Node models which are presented in the next section, and Sensor Node models of NS-2.

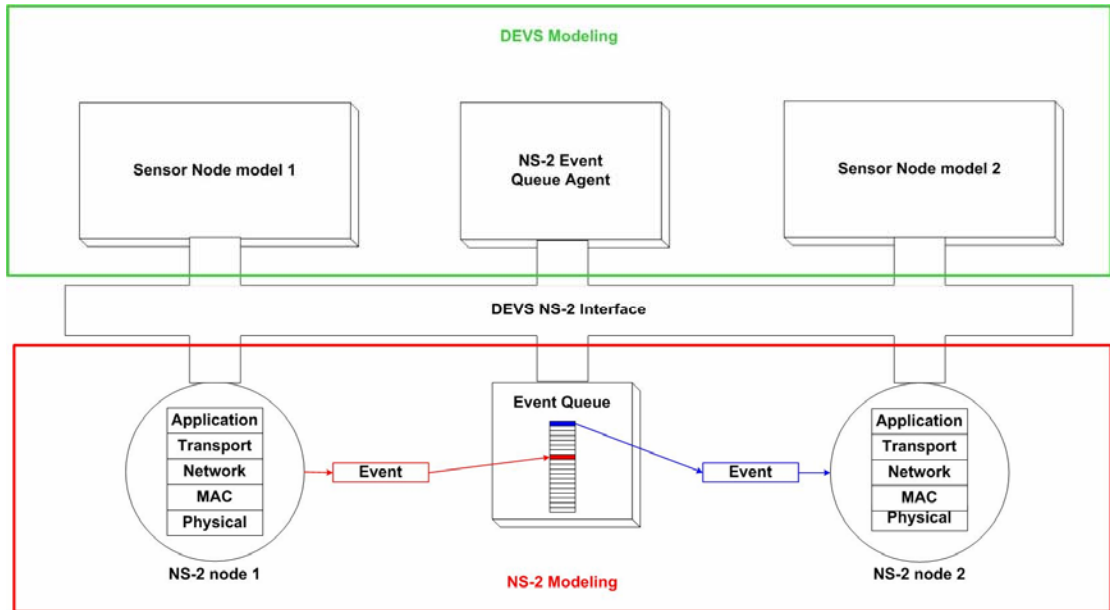


Figure 8. NS-2 Event Queue Agent Model

The time advance ( $ta$ ) of the NS-2 Event Queue Agent model is measured using the information of the first event's scheduled time in the event queue and the current time.

$$ta = \text{first event's scheduled time} - \text{current time}$$

If the NS-2 Event Queue Agent model is imminent (its  $tN = \text{global } tN$ ), then it triggers NS-2 event queue by processing the first event. When  $\delta_{\text{int}}$  is run, this model calls a function of NS-2 in order to dispatch the first event from the event queue and process the event. For example, if a current time is 10 seconds and the first event of the queue is scheduled at 10.3 seconds, then the time advance is set as 0.3 seconds. After 0.3 seconds later, an internal transition function is called. In turn, the first event is

dispatched from the event queue and processed.

### ***3.2. DEVS Model Behaviors***

In this chapter, the simple DEVS atomic models are introduced in order to show that the integration of DEVS and NS-2 has been done without any risk. There are four atomic models and one coupled model. Some of the atomic models are discrete event models and the others are continuous event models. A sensor node model and an NS-2 event queue agent model are discrete event models, but a moving object (tank) model and a collision checker model are continuous event models. In addition, there exists a conceptual interface named as a DEVS NS-2 interface, whose role is to connect DEVS models and NS-2 models. Figure 9 shows the architecture of DEVS models.

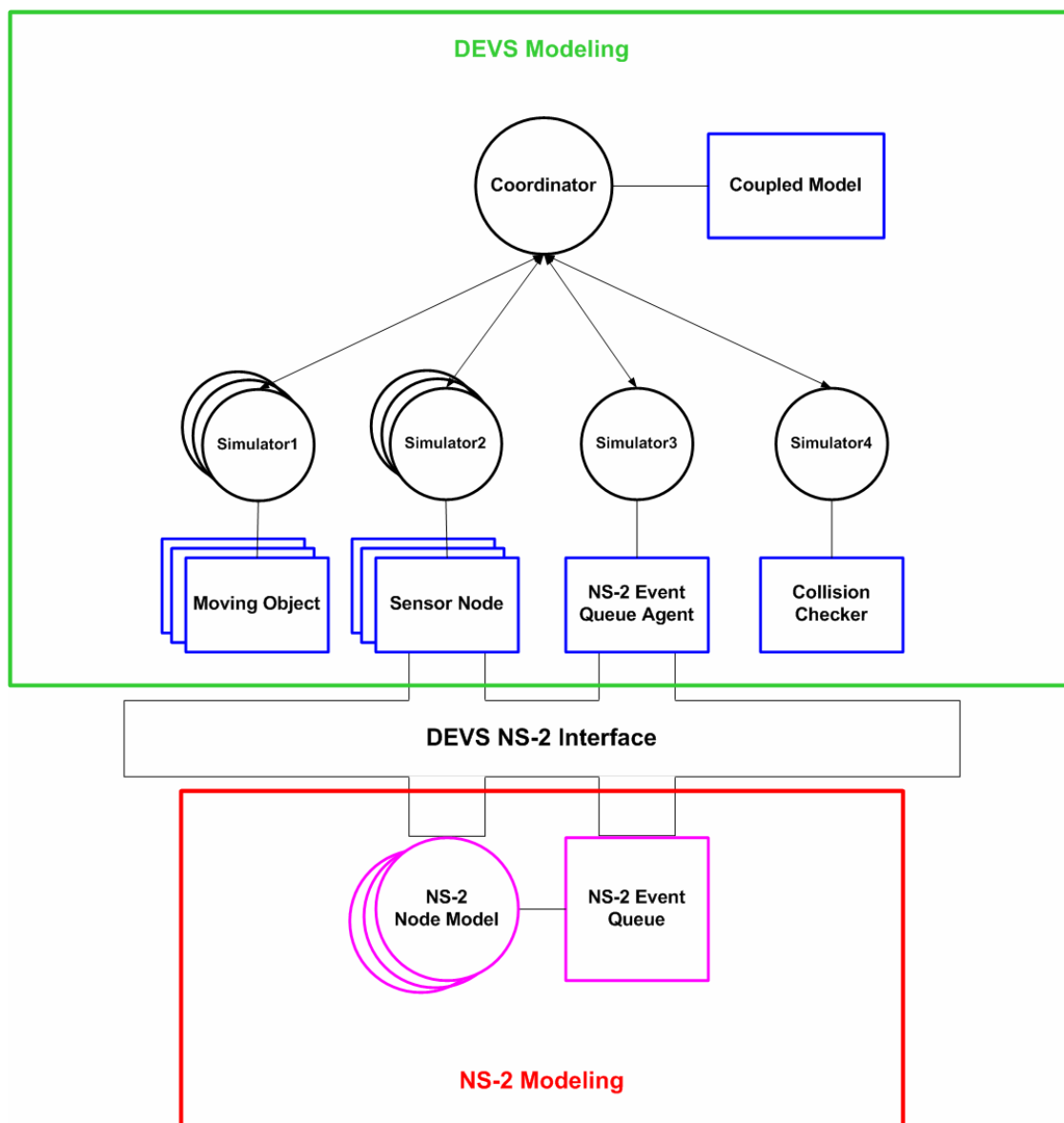


Figure 9. DEVS models in the example of wireless sensor network

The sensor node model has a connection with an NS-2 node model by one-to-one mapping through the DEVS NS-2 Interface. As soon as the DEVS sensor node model detects an event, it triggers its matching NS-2 sensor node to start generating and sending packets. The moving object model such as a tank moves following a pre-defined route,

and the purpose of this model is to give events to the sensor node model. The moving object model is a continuous event model because of its moving behavior. The third atomic model is the NS-2 event queue agent model that is shown in the previous section. Because the NS-2 event queue agent model connects to the NS-2's event queue, this model triggers NS-2 to put events into the event queue whenever a DEVS sensor node model gets events. The last model is the collision checker model. It is a continuous event model and decides whether there are collisions among sensor node models and moving object models. We assume that the word "collision" means sensor node models detect moving objects. Recall that the DEVS NS-2 interface is conceptual. In the last of this chapter, we will discuss in more detail about the DEVS atomic model's behaviors.

### **3.2.1. Sensor Node Atomic Model**

The sensor node model is a fundamental model in a wireless sensor network example. It could be one or more behaviors of an audio sensor, a visual sensor, temperature sensor, etc [15, 16]. Among several kinds of sensors, we assume that visual sensors are used in this thesis. The DEVS sensor node model resides on the top of a NS-2's sensor node model. The DEVS sensor node model represents the behavior of detecting objects and inserting events into the event queues. The role of an NS-2's sensor node model is sending packets. Figure 10 shows the architecture of a sensor node which consists of the DEVS sensor node model and the NS-2's sensor node model.



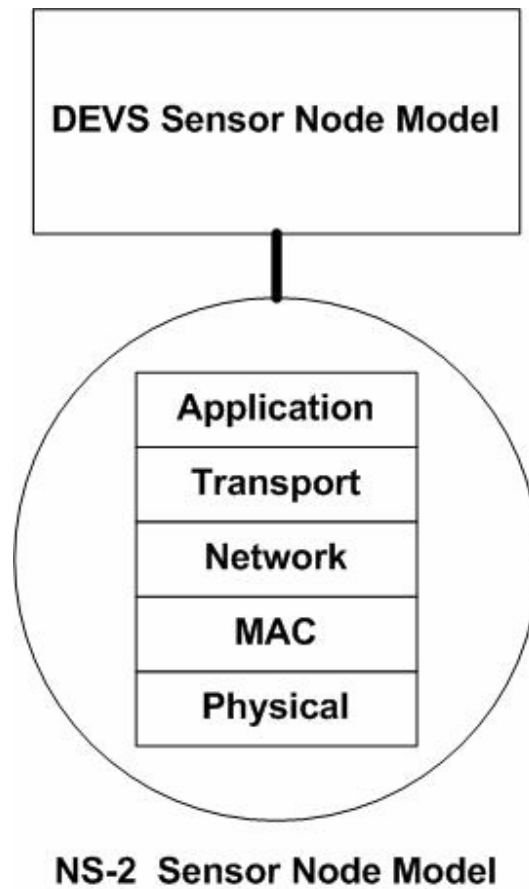


Figure 10. DEVS Sensor Node Model and NS-2 Sensor Node Model

The roles of the DEVS sensor node models and the NS-2 sensor node models are defined. The state transitions are as follows. Initially, the state of a sensor node is “Wait”. It means that the sensor node is sensing an object’s appearance. Once a moving object like a tank comes into a range of the sensor node’s sensing area, the sensor node starts sending packets which say that something has come into their own sensing area. Then, the state of the sensor node becomes the “Generate Packets” state. If the sensor node becomes the “Generate Packets” state, the DEVS sensor node model lets the NS-2 sensor node model, which is the matching model with the DEVS model, generate

packets and send them. During this state, the DEVS sensor node is detecting the moving object which is in its sensing area and the NS-2 sensor node model generates and sends packets continuously with an interval. Once the object disappears from the sensing area of the sensor node, the DEVS sensor node is not able to detect anymore and becomes the “Wait” state. This is the behavior of detecting objects and sending packets. On the other hand, the behavior of a receiver node is between the “Wait” state and the “Receive Packet” state. The state of the sensor node becomes the “Receive Packet” state when the state is “Wait” and the arrived packets destination is itself. Figure 11 shows the behavior of the sensor node model.

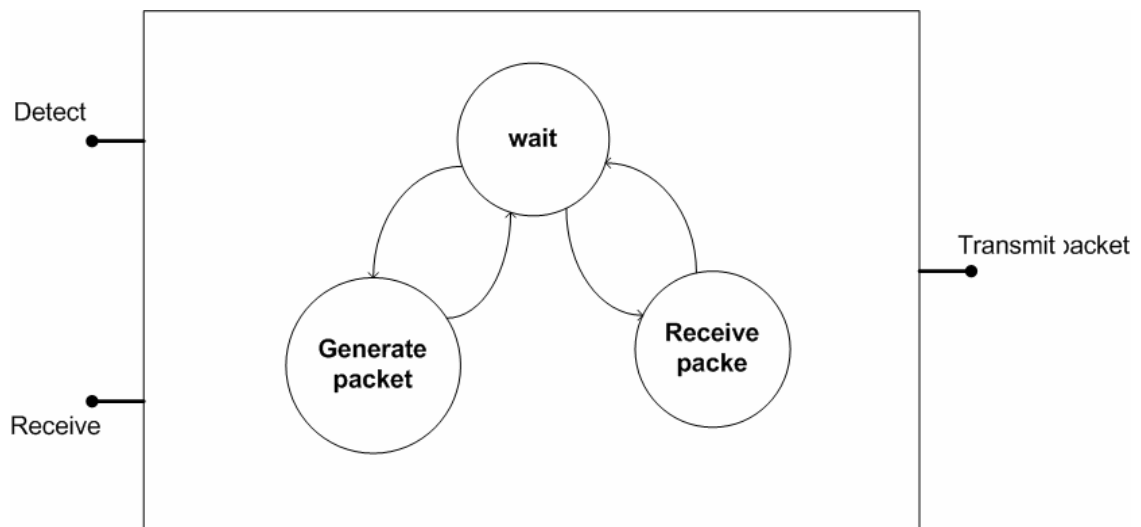


Figure 11. Sensor Node Model State Diagram

There are three states which are “Wait”, “Generate Packet”, and “Receive Packet”. The model has two input ports and one output port. The “Detect” input port

is for sensing moving objects and the “Receive” input port is for receiving packets. The following DEVS formalism represents the sensor node’s behavior that is introduced in above.

$$X = \{ \text{detect, receive} \}$$

$$Y = \{ \text{transmit} \}$$

$$S = \{ \text{wait, generate packet, receive packet} \}$$

$$\delta_{\text{int}} : \text{receive packet} \rightarrow \text{wait}$$

$$\delta_{\text{ext}} : \text{wait} * \text{detect} \rightarrow \text{generate packet}$$

$$\text{generate packet} * \text{detect} \rightarrow \text{wait}$$

$$\text{wait} * \text{receive} \rightarrow \text{receive packet}$$

$$ta : \infty \text{ if } s = \text{wait, generate packet}$$

$$0.001 \text{ if } s = \text{receive packet}$$

The set of input ports is “detect” and “receive”. There is one output port which is “transmit”. The states of the model are “wait”, “generate packet”, and “receive packet”. The internal state transition function ( $\delta_{\text{int}}$ ) changes the state of “receive packet” to “wait” after time advance (ta). The external state transition function ( $\delta_{\text{ext}}$ ) has 3 roles. If a sensor node gets an input event from the “detect” input port during the “wait” state, it becomes the “generate packet” state. In another case, a sensor node is

able to receive events from the “detect” input port during the “generate packet” state. Input events, which come into the “detect” input port during the “generate packet” state, mean that moving objects that have been detected get out of a sensing area of a node. Then, the state becomes the “wait” state. The last role of the external state transition function is to change the “wait” state to the “receive packet” state when a sensor node receives input events from the “receive” input port. The time advance (ta) is set to infinity when the “wait” or the “generate packet” states and the internal transition function never happens. Those two states can be changed to the other states only if a sensor node gets input events. Otherwise, the time advance is set as 0.001 when the state is “receive packet”. A received packet processing time is assumed as 0.001 second. That’s all about the design consideration of the DEVS sensor node model.

### 3.2.2. Moving Object(Tank) Atomic Model

The DEVS moving object model which has simple moving behavior is implemented. Figure 12 depicts the state transition diagram of the DEVS moving object model.

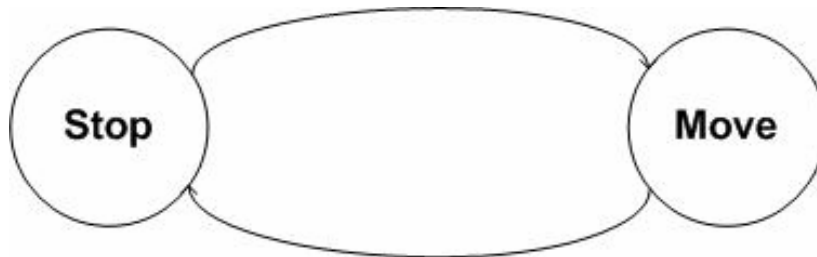


Figure 12. Moving Object(Tank) Model State Diagram

The DEVS moving object model has two states of “stop” and “move”. Once a simulation starts, the model is set as the “move” state initially and starts moving from an original position to a destination position. A moving route is pre-decided when a moving object model is created. The behavior of a moving and a current position vector are defined below.

$$\vec{C} = \vec{O} + (\vec{D} - \vec{O}) * \lambda$$

$$\text{where, } 0 \leq \lambda \leq 1$$

$$\lambda = \frac{e}{ta}$$

$$0 \leq \frac{e}{ta} \leq 1$$

$$\text{So, } \vec{C} = \vec{O} + (\vec{D} - \vec{O}) * \frac{e}{ta}$$

$\vec{C}$  : *Current Position*

$\vec{O}$  : *Original Position*

$\vec{D}$  : *Destination Position*

We assumed time advance (ta) as required time for a moving object from one point to another point. A phi function is defined with the above vector equation of calculating a current position. The reason why a phi function is used is that the moving behavior of a moving object (tank) is continuous. The phi function depicts a continuous behavior in DEVS formalism. The moving area is limited in two dimensional space in this thesis. The scenario that will be discussed in chapter 3.3 is done in 100\*100 size

space. Initially, the position of a moving object is set as (0,0), and it starts moving to a destination point once a simulation starts. During time advance, a moving object is heading to a destination point and reaches its destination at the time of time advance. The current position during moving can be calculated with the original position, the destination position,  $t_a$ , and elapsed time ( $e$ ). If a moving object arrives at a destination position, a delta external function is called and a moving object changes its direction according to the next destination. Then, a starting position, a current position, and a new destination position are updated. During simulation, a moving object moves continuously for sensor nodes to detect itself in order to measure the energy consumption.

### 3.2.3. Collision Checker Atomic Model

Each sensor checks if there is a moving object in its sensing area continuously. The DEVS collision checker model is responsible for checking whether sensor nodes detect moving objects. The shorter sensing interval, the more accurate and more computational power required. The sensing interval is set as 0.1 second in this thesis. In order to set the sensing interval time, the phi function is called every 0.1 second. Figure 13 shows the state diagram of the DEVS collision checker model.

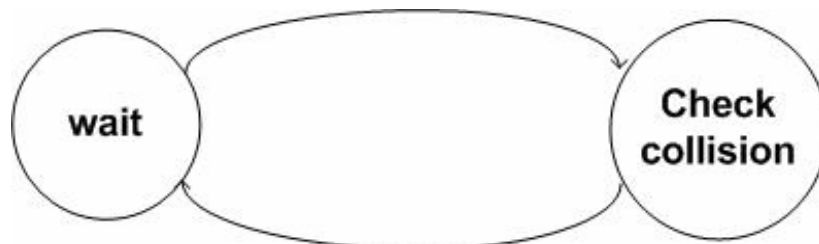


Figure 13. Collision Checker Model State Diagram

This model has two states of “wait” and “check collision”. This model is keeping two lists. One is a list of sensor nodes and the other is a list of moving objects. At every interval, this model measures the distance between moving object models and sensor nodes. If a moving object is in the sensing area of a sensor node, the collision checker model sends a message to the sensor node’s input port of “detect”. The next process is followed by a sensor node model’s behavior that we discussed in chapter 3.2.1.

### ***3.3. Scenario for confirming the integration***

To simulate the scenario for the integration of DEVS and NS-2, a node configuration needs to be defined first, and a network topology is required to be built, including sensor nodes, a base node, and a moving object(tank) in the topology. At last the simulation result is gained and compared with the result of stand alone NS-2 simulation. If two results of both a DEVS-NS2 simulation and a stand alone NS-2 simulation are the same or very similar, the integration of DEVS and NS-2 is considered successful.

#### **3.3.1. Node Configuration**

To simulate the scenario for the integration of DEVS and NS-2, a node configuration needs to be defined. An NS-2 sensor node configuration is defined in Tcl

script code. Node configuration consists of defining the different node protocol characteristics before creating them. They may consist of the type of addressing structure used in the simulation, defining the network components for models, selecting the type of a routing protocol, and defining the energy model [17, 18, 19]. The node configuration command looks like Figure 14.

set val(chan)	Channel/WirelessChannel	;# channel type
set val(prop)	Propagation/TwoRayGround	;# radio-propagation model
set val(netif)	Phy/WirelessPhy	;# network interface type
set val(mac)	Mac/802_11	;# MAC type
set val(ifq)	Queue/DropTail/PriQueue	;# interface queue type
set val(ll)	LL	;# link layer type
set val(ant)	Antenna/OmniAntenna	;# antenna model
set val(ifqlen)	50	;# max packet in ifq
set val(rp)	DumbAgent	;# routing protocol
set opt(engmodel)	EnergyModel	;# energy model
set opt(initeng)	100.0	;# Initial energy in Joules

```
$ns_ node-config -adhocRouting $val(rp) \
                -llType $val(ll) \
                -macType $val(mac) \
                -ifqType $val(ifq) \
```



```

-ifqLen $val(ifqlen) \
-antType $val(ant) \
-propType $val(prop) \
-phyType $val(netif) \
-channelType $val(chan) \
-topoInstance $wtopo \
-initialEnergy $opt(initeng) \
-energyModel $opt(engmodel) \
-agentTrace ON \
-routerTrace ON \
-macTrace OFF \
-movementTrace OFF \

```

Figure 14. Node Configuration

In this example, node configuration for a wireless sensor node assigns the Dumb Agent routing algorithm as its adhoc routing protocol. The link layer type as LL and Medium Access Control(MAC) protocol as IEEE 802.11 are configured. The queue between the MAC layer and the link layer is used. At last, the wireless physical layer, if defined. The network stack is configured by assigning the protocols for the link layer, the MAC layer, and the physical layer. Because the sensor nodes communicate with each other using a wireless network, a channel topology and a propagation model are needed. Consequently, the Omni Antenna, the wireless channel, and the two-lay ground

propagation are assigned in the node configuration. The initial energy model is set as 100 joules in order to measure the energy consumption. The sensor nodes are assumed as they don't move. The router trace and the agent trace are turned on to investigate the simulation results. This procedure creates the sensor node object, creates an adhoc-routing agent as specified, and creates the network stack consisting of the link layer, interface queue between the link layer and the MAC layer, the MAC layer, and the network interface with the antenna. This procedure also uses the defined propagation model, interconnects these components and connects the stack to the schematic in Figure 15. Figure 15 shows the schematic of the NS-2 sensor node's layered network protocol.

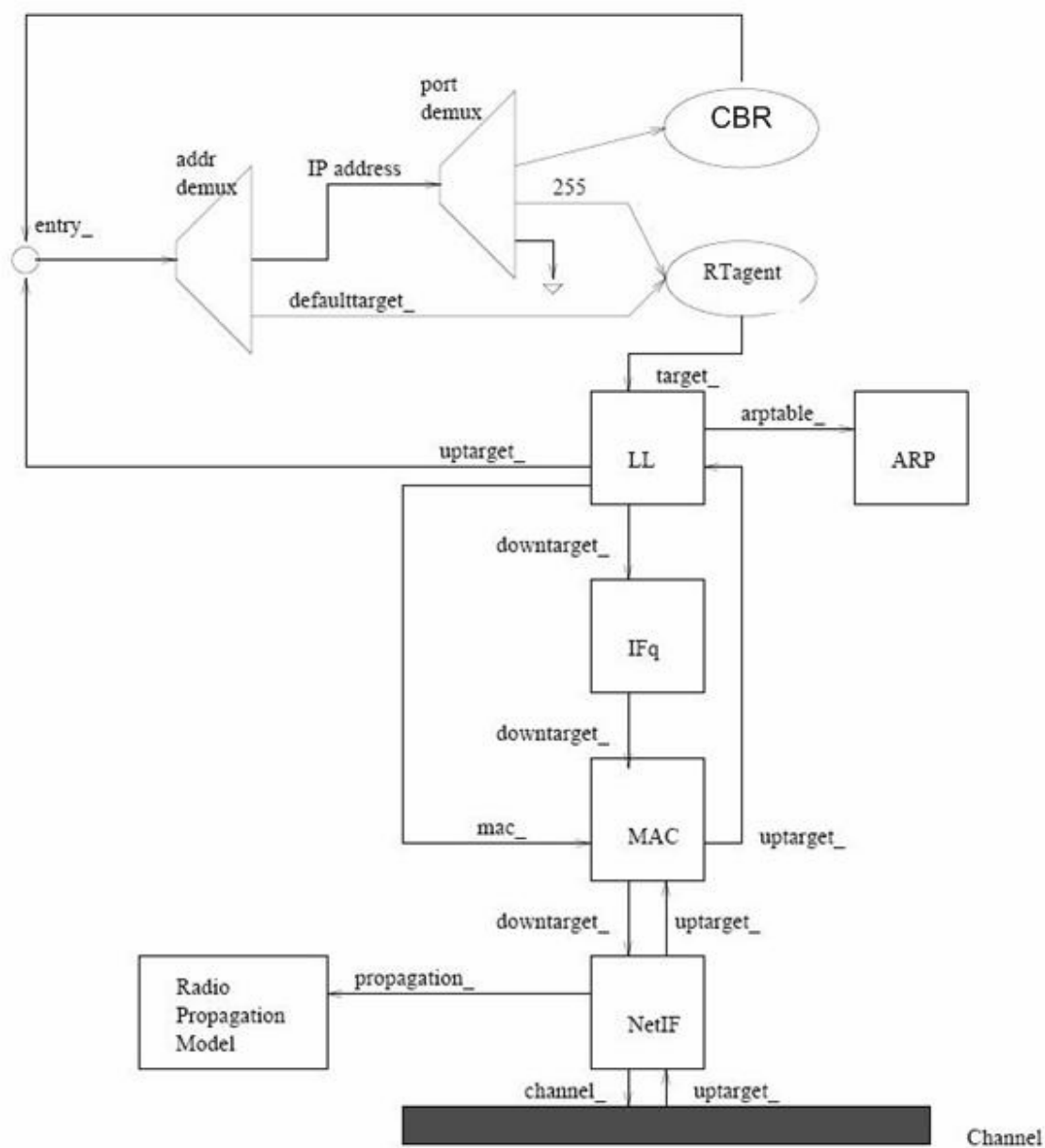


Figure 15. Schematic of the Sensor Node

A Constant Bit Rate (CBR) traffic generator is assigned for the application agent. In the DEVS-NS2, a DEVS model conceptually resides on the top of an NS-2 model such

as Figure 10. As a result, the DEVS sensor node model and the NS-2 sensor node model can communicate with each other by passing messages. The DEVS sensor node model doesn't have to know the details of the NS-2 sensor node model's configuration and how it works. What the DEVS sensor node model needs to do is sending event messages to the NS-2 sensor node model when the DEVS sensor node model gets events. In this example, the DEVS sensor node model is connected with its own NS-2 sensor node model's CBR traffic generator agent that is used for the application of the sensor node. The DEVS sensor node model considers only CBR traffic generator as its own NS-2 node model, but the lower layers of the NS-2 sensor node are abstracted. Figure 16 shows the relation of the DEVS sensor node model and the NS-2 sensor node model.

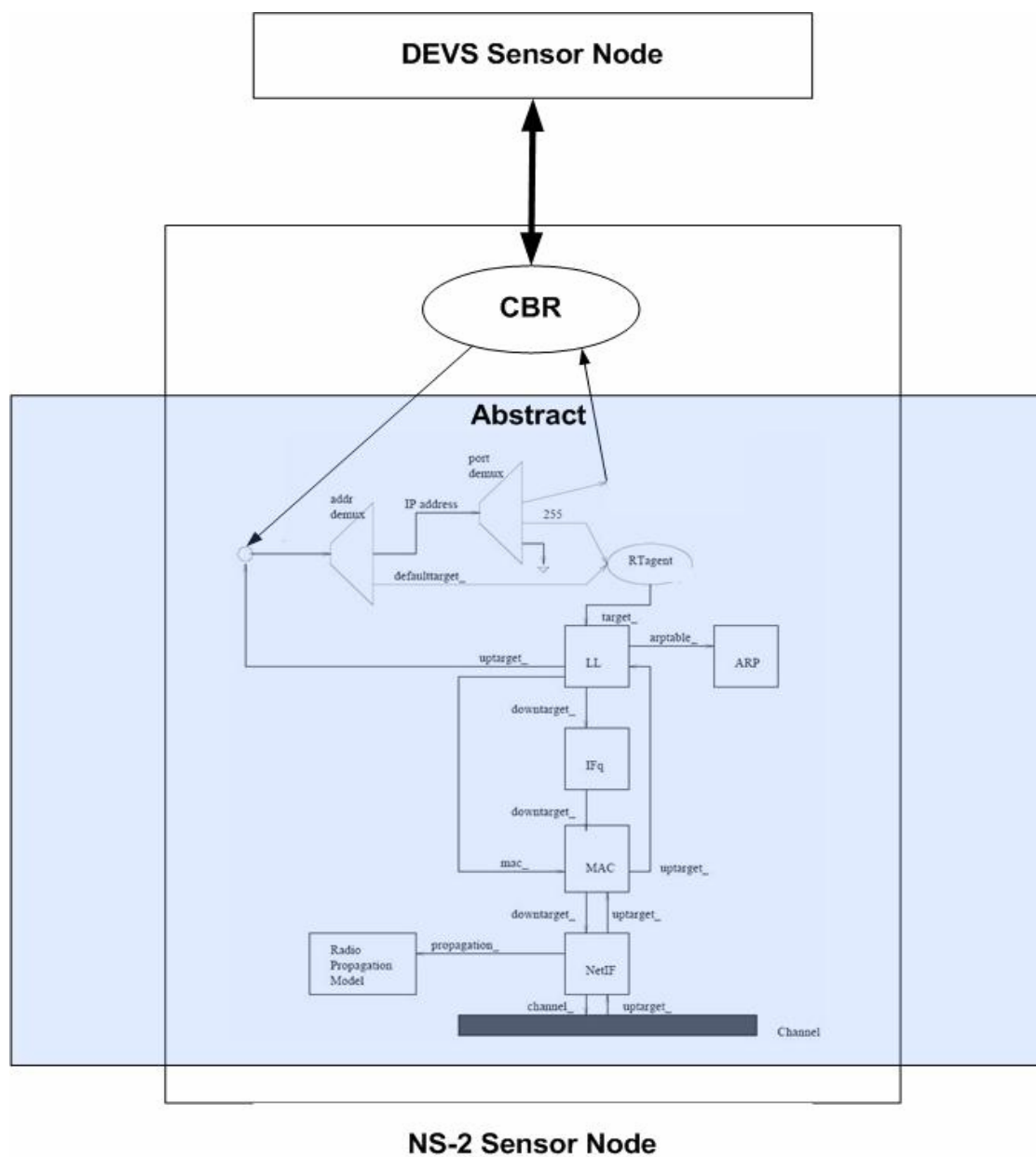


Figure 16. Schematic of the relation with DEVS and NS-2 Sensor Node

### 3.3.2. Network Topology

In this section, the simple network topology is presented to get the simulation result of energy consumptions and packet deliveries. Seven sensor nodes, one base node, and one tank (moving object) are initialized in a 100\*100 space. The detailed behavior of the sensor node, the base node, and the moving object is explained in the section 3.2. Figure 17 shows that the network topology and the route for tank.

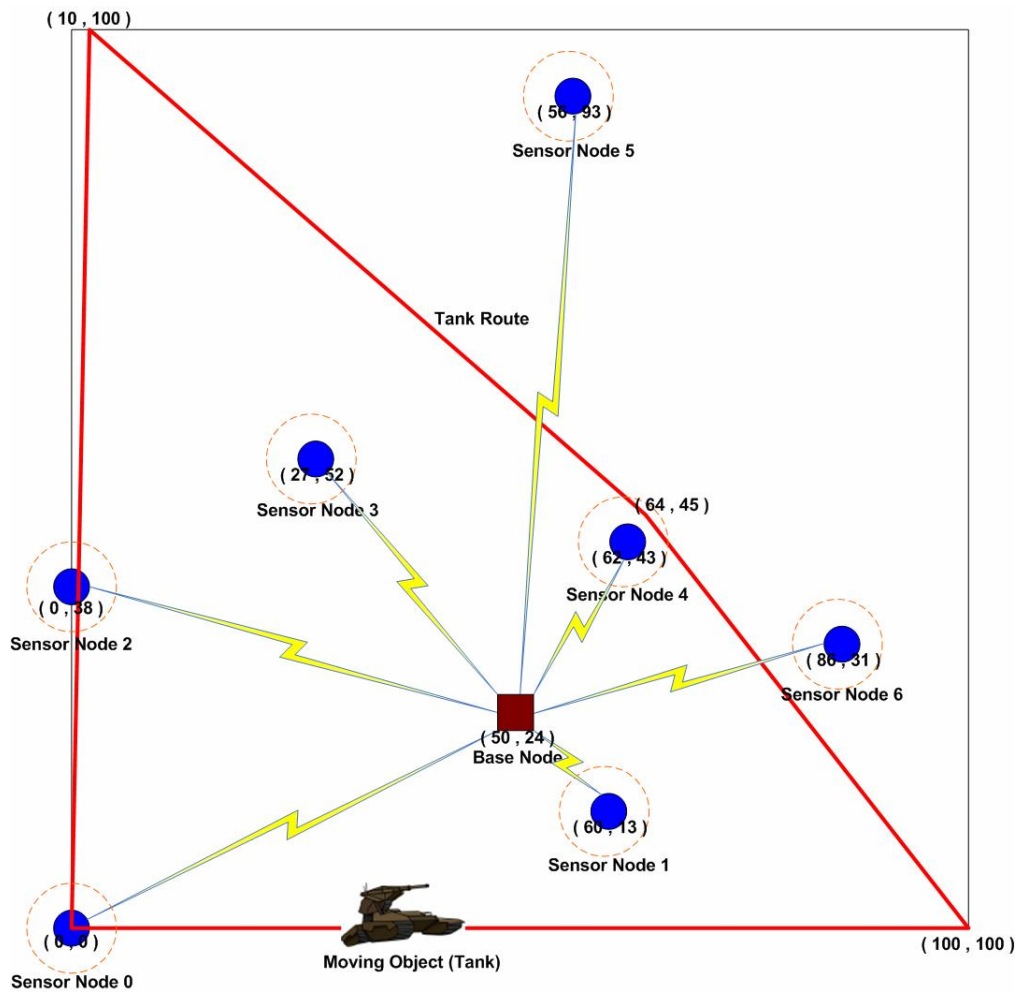


Figure 17. Network Topology and pre-defined tank route

In Figure 17, the filled circles represent sensor nodes and dot circle lines mean sensing areas of each sensor node. A base station node is represented as the filled square. There is one moving object (tank) and a pre-defined tank route. All the sensor nodes and the base station node communicate through a radio based wireless network. The data packets are delivered from the sensor nodes to the base node using the NS-2 simulator and the NS-2 layered network objects. Because the DEVS node model resides on the top of the NS-2 node, the DEVS models don't have to consider the packet delivery but instead deals with the detection of moving objects. The sensor node 0, 1, 2, 3, 4, 5, and 6 are able to detect the tank's movement. They send packets as soon as they sense the tank with every DEVS collision checker model's checking interval which is represented in the section 3.2.3. The base node, which is the destination node of all the sensor nodes, receives packets from the sensor nodes. When the base node receives packets, the base node discards the packets. In turn, it is ready to receive other packets. In the following section, the simulation results are discussed.

### **3.3.3. Simulation Results**

Figure 18 shows the simulation console which is initialized and ready to run. Seven sensor nodes (node 0, 1, 2, 3, 4, 5, and 6), one base node (node 7), and one moving object (tank) are initialized and wait to be run. The nodes' positions are set like the network topology in Figure 17.

```

TaekyuKim@taekyu ~/.ns-allinone-2.28/my_example/scenario1
$ ns s1_ex.tcl
num_nodes is set 8
warning: Please use -channel as shown in tcl/ex/wireless-mit
f.tcl
INITIALIZE THE LIST xListHead
Loading connection pattern...
Starting Simulation...

~~~~~DEUS Engine starts~~~~~
No of G_DvNSCBB_Mnrg = 8
n0 = < 0.000000 , 0.000000 , 0.000000 >
n1 = < 60.000000 , 13.000000 , 0.000000 >
n2 = < 0.000000 , 38.000000 , 0.000000 >
n3 = < 27.000000 , 52.000000 , 0.000000 >
n4 = < 62.000000 , 43.000000 , 0.000000 >
n5 = < 56.000000 , 93.000000 , 0.000000 >
n6 = < 86.000000 , 31.000000 , 0.000000 >
n7 = < 50.000000 , 24.000000 , 0.000000 >

No of MOVING_OBJECT = 1
No of SENSOR_NODE = 7
No of BASE_NODE = 1

Type 'SimStart', 'SimEnd' : _

```

Figure 18. Simulation Initialization

When the simulation starts, the tank, whose current position is (0,0), moves toward the next destination (100,0) such as we illustrate in Figure 17. As soon as the tank starts moving, the sensor node 0, located in (0,0) detects the tank and starts generating packets and sends them to the base node (node 7). If the tank is out of the sensing range of the sensor node 0, the sensor node 0 stops generating packets. The tank moves toward the four points which are (100,0), (64,45), (10,100), and (0,0).



Every time the tank reaches the point, the starting point and the destination point are updated, and the tank continues to move following the route. The moving object model calculates its position in regards to the distance from the starting point and the destination point with the time advance and the elapsed time. Because every route has a different distance, the tank's speed varies. During one cycle of the tanks movement, three sensor nodes which are node 0, node 2, and node 4 detect the tank and send packets to the base node while the tank is in their sensing area. The sensor nodes' detecting times during 3 cycles are attained through the simulation. Table 1 shows the times for the sensor nodes to detect the tank.

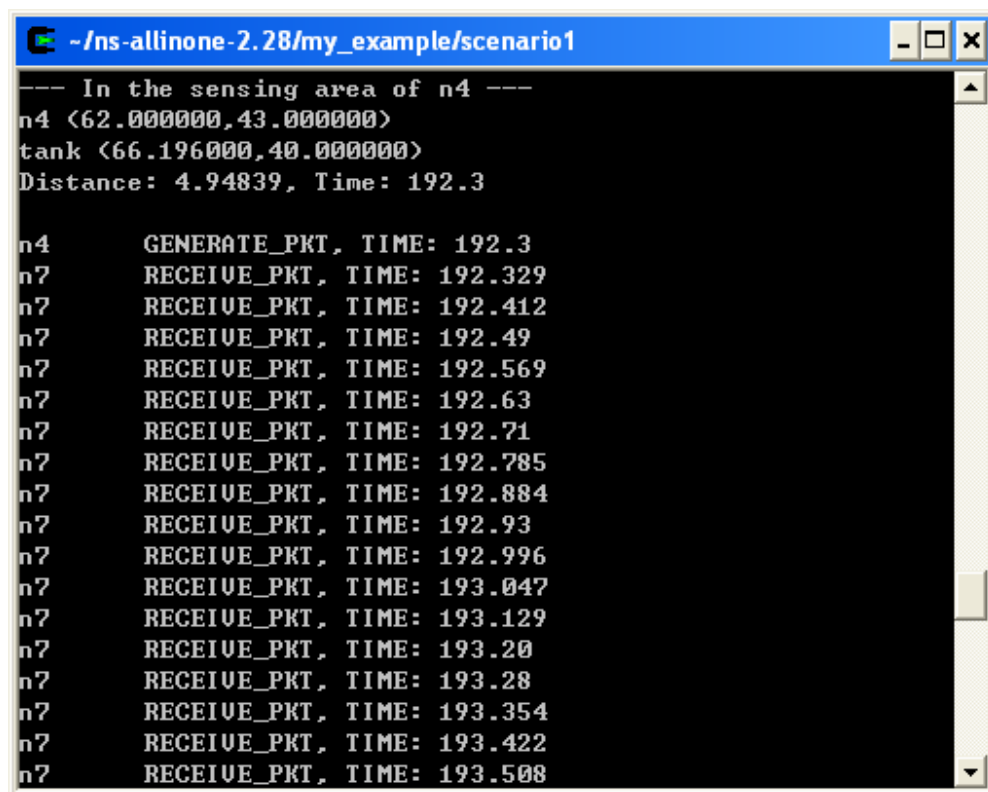
Cycle	Time(sec)	Sensor Node 0	Sensor Node 2	Sensor Node 4
1	0.1	Generate		
	5.1	Wait		
	192.3		Generate	
	205.4		Wait	
	359.2			Generate
	365.7			Wait
2	395.1	Generate		
	405.1	Wait		
	592.3		Generate	
	605.4		Wait	

	759.2			Generate
	765.7			Wait
3	795.1	Generate		
	805.1	Wait		
	992.3		Generate	
	1005.4		Wait	
	1159.2			Generate
	1165.7			Wait

Table 1. The simulation results for the sensor nodes to detect the tank

In Table 1, the sensor nodes achieve the “Generate” state when the sensor nodes detect the tank and send packets to the base node until the tank disappears out of the sensing area of the sensor nodes and the sensor nodes become the “Wait” state to stop generating packets. Between the times of the “Generate” and the “Wait” states, each sensor node generates packet and consumes the energy. At the same time the base node receives the packets from the sensor node with some delay and the base node also needs to consume the energy in order to receive packets. NS-2 deals with how much of the energy is required and how long it takes for packets to be delivered from the source node to the destination node. The energy consumption for each node and the total number of packets generated in the whole network topology are measured after the simulation that is 1165.7 second long. Figure 19 is the simulation console that shows a sensor node’s

behavior and a base node's behavior. When a sensor node detects a tank in its sensing area, a sensor node becomes the "Generate" state and sends packets with a transmit interval time that is assigned in Tcl script code.



```

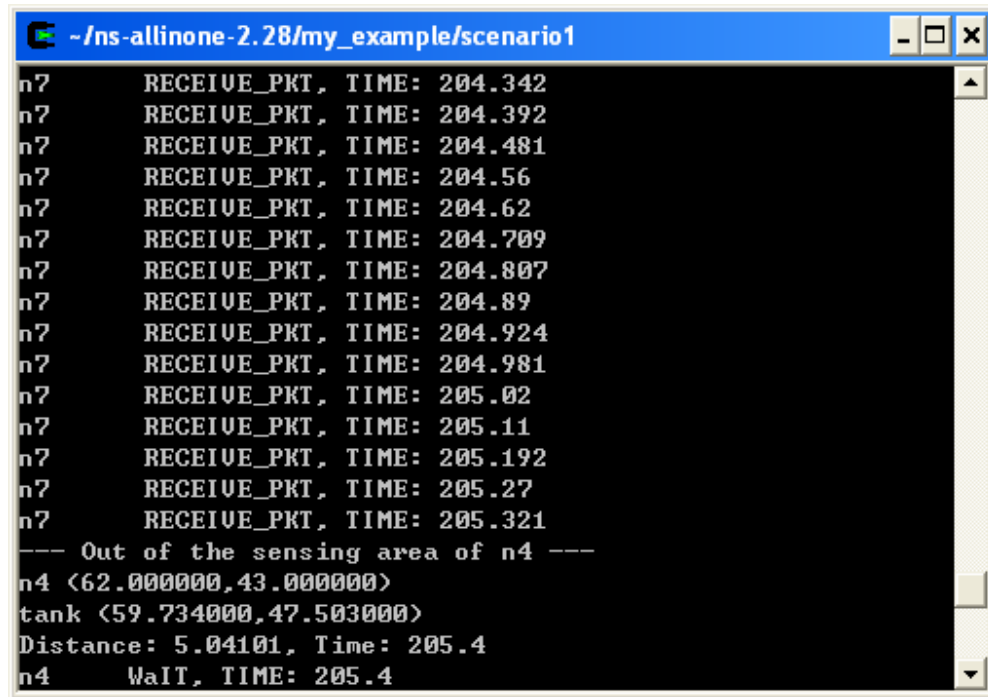
--- In the sensing area of n4 ---
n4 <62.000000,43.000000>
tank <66.196000,40.000000>
Distance: 4.94839, Time: 192.3

n4      GENERATE_PKT, TIME: 192.3
n7      RECEIVE_PKT, TIME: 192.329
n7      RECEIVE_PKT, TIME: 192.412
n7      RECEIVE_PKT, TIME: 192.49
n7      RECEIVE_PKT, TIME: 192.569
n7      RECEIVE_PKT, TIME: 192.63
n7      RECEIVE_PKT, TIME: 192.71
n7      RECEIVE_PKT, TIME: 192.785
n7      RECEIVE_PKT, TIME: 192.884
n7      RECEIVE_PKT, TIME: 192.93
n7      RECEIVE_PKT, TIME: 192.996
n7      RECEIVE_PKT, TIME: 193.047
n7      RECEIVE_PKT, TIME: 193.129
n7      RECEIVE_PKT, TIME: 193.20
n7      RECEIVE_PKT, TIME: 193.28
n7      RECEIVE_PKT, TIME: 193.354
n7      RECEIVE_PKT, TIME: 193.422
n7      RECEIVE_PKT, TIME: 193.508

```

Figure 19. Simulation Console when a sensor node detects a tank

Figure 20 shows the case which a tank disappears from a sensor node's sensing area. When a sensor node is not able to detect a tank in its sensing area and its state is "Generate", an a sensor knows that a tank goes out of the sensing area, the sensor node attains the "Wait" state and stays in the "Wait" state until it detects something again.



```

- /ns-allinone-2.28/my_example/scenario1
n7 RECEIVE_PKT, TIME: 204.342
n7 RECEIVE_PKT, TIME: 204.392
n7 RECEIVE_PKT, TIME: 204.481
n7 RECEIVE_PKT, TIME: 204.56
n7 RECEIVE_PKT, TIME: 204.62
n7 RECEIVE_PKT, TIME: 204.709
n7 RECEIVE_PKT, TIME: 204.807
n7 RECEIVE_PKT, TIME: 204.89
n7 RECEIVE_PKT, TIME: 204.924
n7 RECEIVE_PKT, TIME: 204.981
n7 RECEIVE_PKT, TIME: 205.02
n7 RECEIVE_PKT, TIME: 205.11
n7 RECEIVE_PKT, TIME: 205.192
n7 RECEIVE_PKT, TIME: 205.27
n7 RECEIVE_PKT, TIME: 205.321
--- Out of the sensing area of n4 ---
n4 (62.000000,43.000000)
tank (59.734000,47.503000)
Distance: 5.04101, Time: 205.4
n4 Wait, TIME: 205.4

```

Figure 20. Simulation Console when a tank disappears from the sensing area

Figure 19 and the Figure 20 represent the behaviors of sensor node 4 and the base node. The sensor node 0 and 2 also work as similar as the sensor node 4 does according to the tank's movement. Figure 21 and Figure 22 represent that the simulation results in terms of the energy consumption for each node and the total number of packets that are generated in the whole network topology. In this simulation, the initial energy is set to 100 joules for each node. Nodes spend energy when they send, receive, or transmit packets. Figure 21 is the chart that shows the remaining energy in each node. Although there are no packets generated in sensor nodes 1, 3, 5, and 6, they consume energy because they receive packets anyway and discard them because they are

not the destination for the packets.

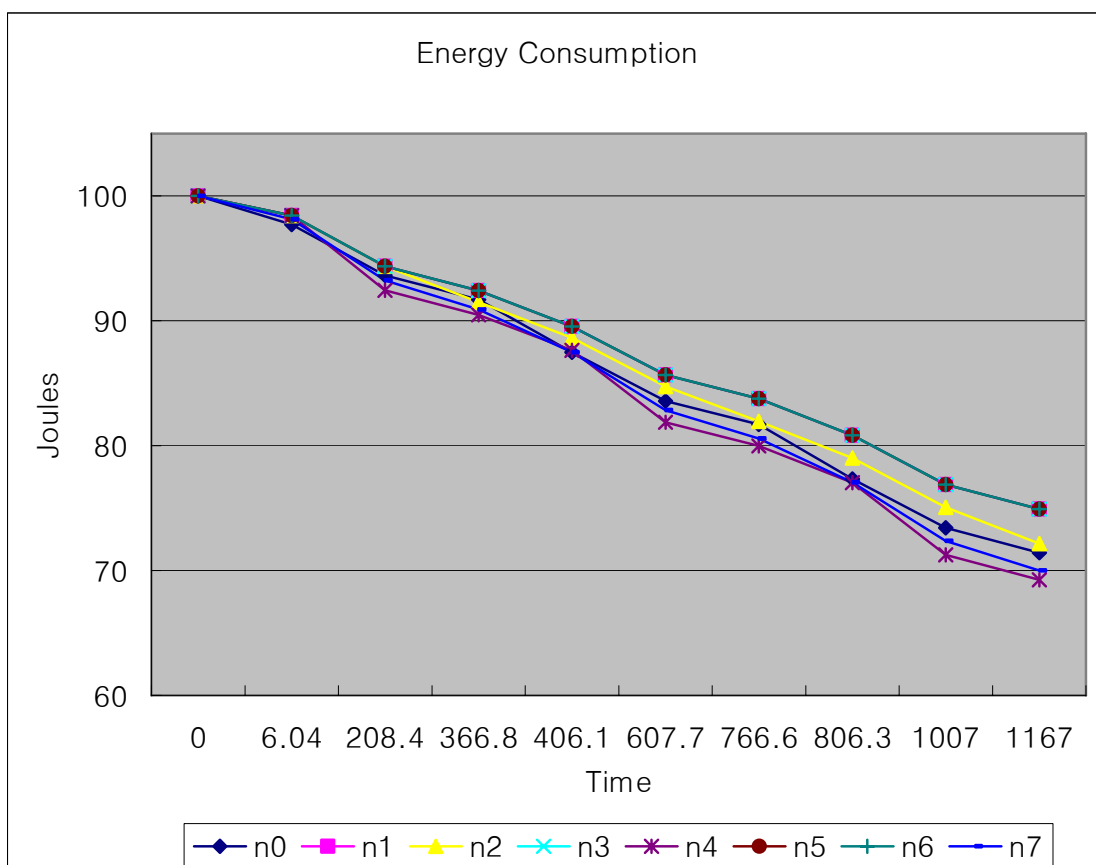


Figure 21. The remaining energy for each node

Table 2 depicts the energy remaining during the simulation process. Every node starts with 100 joules energy and the nodes finish the simulation with different amount of energy. The energy consumption depends on how many packets a node generates, receives, or transmits.

Time	Energy Remains (Joules)							
	n0	n1	n2	n3	n4	n5	n6	n7
0	100	100	100	100	100	100	100	100
6.04	97.72	98.44	98.44	98.44	98.44	98.44	98.44	98.13
208.37	93.65	94.37	94.37	94.37	92.44	94.37	94.37	93.25
366.75	91.69	92.41	91.49	92.41	90.49	92.41	92.41	90.90
406.1	87.48	89.56	88.64	89.56	87.64	89.56	89.56	87.49
607.67	83.58	85.66	84.74	85.66	81.87	85.66	85.66	82.83
766.58	81.68	83.77	81.95	83.77	79.98	83.77	83.77	80.58
806.28	77.34	80.83	79.00	80.83	77.04	80.83	80.83	77.06
1007.49	73.41	76.89	75.07	76.89	71.23	76.89	76.89	72.37
1167.24	71.43	74.9	72.14	74.91	69.25	74.91	74.91	70.00

Table 2. The energy remains during the simulation process

The Figure 22 represents the total number of packets that are generated at every node during the simulation. The total number of the generated packets is measured at the end of the third cycle of the tank's movement. 829 packets have been generated during the simulation.

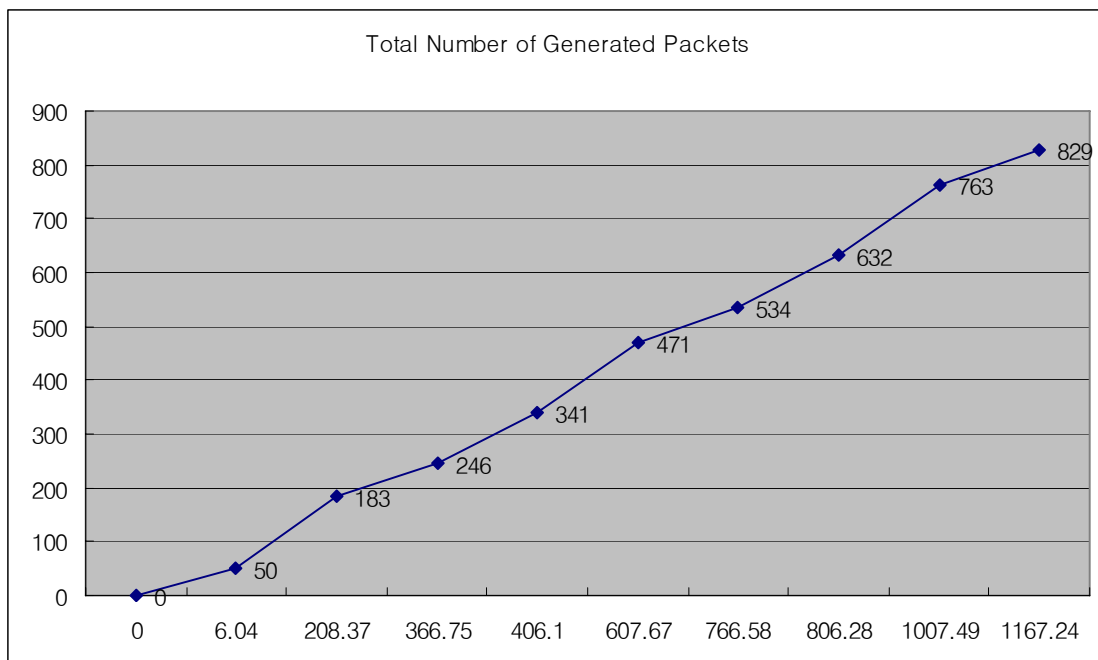


Figure 22. The total number of generated packets in the simulation.

The Figure 23 shows that the number of packets generated and the energy consumption of the sensor nodes 0, 2, and 4 which are the nodes that detect the tank and generate packets. The results are reasonable since generating more packets leads to more energy consumption.

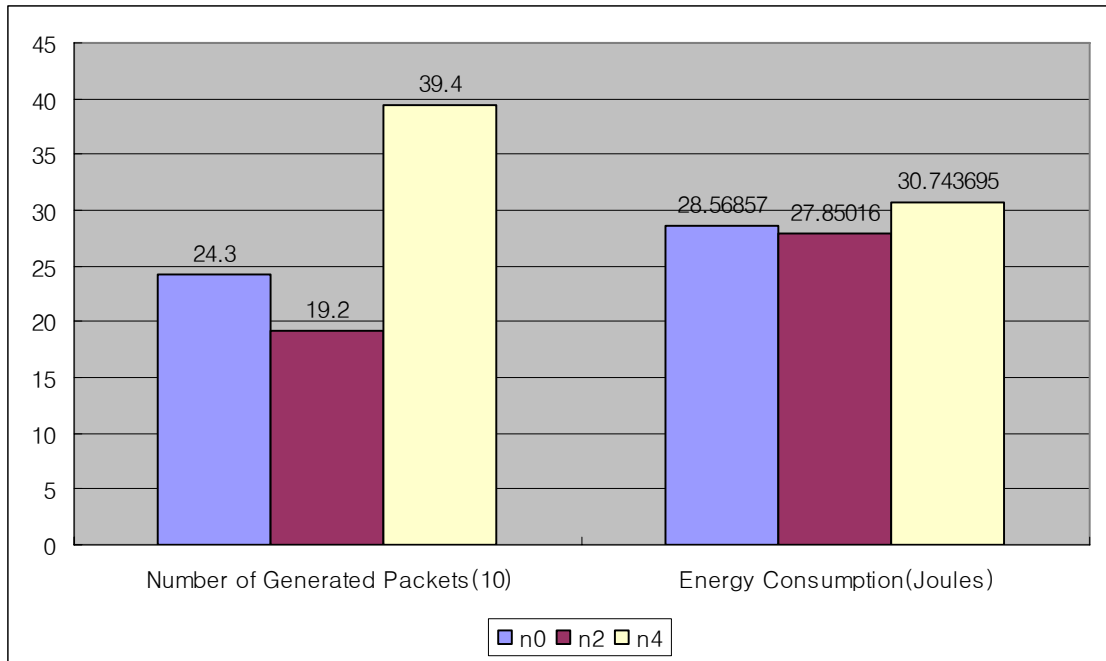


Figure 23. The simulation result at the sensor node 0, 2, 4

The Figure 21, 22, and 23 depict the results of the DEVS-NS2 simulation. Now, the simulation with NS-2 alone and the simulation results are needed in order to compare with the results which are obtained by the DEVS-NS2 simulation. To make same environment, the pre-defined scenario is batched in Tcl script code. Instead of the behavior of a tank and the behavior of a sensor node, generating packet times and stopping packet time are assigned to the sensor nodes 0, 2, and 4 such as the results of the DEVS-NS2 simulation shown in Table 1. The pre-defined scenario in Tcl script codes are shown in Figure 24.

```
### First Cycle ###
```



\$ns_ at 0.1	"\$cbr_(0) start"
\$ns_ at 5.1	"\$cbr_(0) stop"
\$ns_ at 192.3	"\$cbr_(4) start"
\$ns_ at 205.4	"\$cbr_(4) stop"
\$ns_ at 359.2	"\$cbr_(2) start"
\$ns_ at 365.7	"\$cbr_(2) stop"
### Second Cycle ###	
\$ns_ at 395.1	"\$cbr_(0) start"
\$ns_ at 405.0	"\$cbr_(0) stop"
\$ns_ at 592.3	"\$cbr_(4) start"
\$ns_ at 605.4	"\$cbr_(4) stop"
\$ns_ at 759.2	"\$cbr_(2) start"
\$ns_ at 765.7	"\$cbr_(2) stop"
### Third Cycle ###	
\$ns_ at 795.1	"\$cbr_(0) start"
\$ns_ at 805.0	"\$cbr_(0) stop"
\$ns_ at 992.3	"\$cbr_(4) start"
\$ns_ at 1005.4	"\$cbr_(4) stop"
\$ns_ at 1159.2	"\$cbr_(2) start"
\$ns_ at 1165.7	"\$cbr_(2) stop"

Figure 24. Pre-defined scenario code

With this pre-defined scenario and the same network topology which is represented in Figure 17, the NS-2 alone simulation is processed. The results regarding the energy consumption and the number of packets generated are exactly the same as the DEVS-NS2 simulation results. So, it is concluded that the integration of DEVS and NS-2 is accomplished risk free.

### ***3.4. Reference Comparison***

There is a previous study that is similar to DEVS-NS2. The previous study presented a heterogeneous simulation framework using DEVS BUS [6]. The purpose of the DEVS BUS framework is to interoperate a collection of simulators which are developed in different environments such as DEVS, NS, and C++Sim [20]. The DEVS BUS framework is recapitulated first and, in turn, the DEVS-NS2 is compared with the DEVS BUS framework so that advantages and disadvantages between them are found out.

#### **3.4.1. DEVS BUS framework**

In this section, the DEVS BUS framework is briefly introduced. The main goal of the DEVS BUS framework is to achieve a unified simulation infrastructure with different simulation protocols. There are two main concepts to integrate existing simulation models. One is the simulation protocol conversion and the other is the DEVS BUS controller. Figure 25 shows the basic conceptual architecture of the DEVS BUS framework.

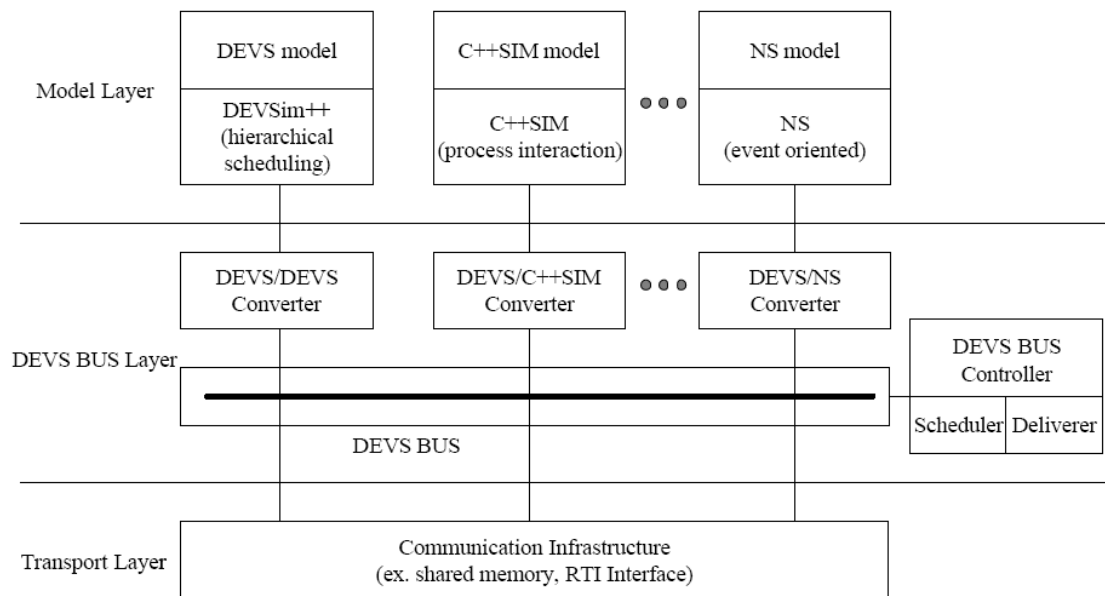


Figure 25. DEVS BUS framework [6]

The DEVS BUS framework consists of three layers: the model layer, the DEVS BUS layer, and the transport layer. The model layer has different simulation models such as DEVS, C++Sim, and NS model. The second layer is the DEVS BUS layer which mainly deals with heterogeneous simulation. The DEVS BUS is a virtual software bus that performs the two functions of time synchronization and message passing. Two components exist in this layer: the protocol converter and the DEVS BUS controller. The protocol converter changes non-DEVS models into DEVS models so that time synchronization and message passing can be attained. The DEVS BUS controller has a centralized scheduler which keeps the global simulation time in order for nodes to advance their local time after receiving a time advance from the scheduler. The

DEVS BUS also contains a deliverer which stores all connection information among simulation models and supervises every message passing between nodes. The method of using the protocol converter and the controller makes time synchronization easy and increases modeling flexibility. However, a model sends messages to a destination model through a centralized deliverer in DEVS BUS framework. So, every message is delivered to destinations through a centralized deliverer. As a result, this message passing mechanism results in a bottleneck and decreases a computational speed. The lowest layer is the transport layer which supports communication between models. Any kind of communication infrastructure such as shared memory, TCP/IP, or HLA/RTI can be used. This is the summary of the DEVS BUS framework. The DEVS-NS2 is compared with the DEVS BUS framework in the next section.

### **3.4.2 Comparison between DEVS-NS2 and DEVS BUS framework**

The DEVS-NS2 is implemented for integrating DEVS models only into NS-2 models. However, the DEVS BUS framework is more flexible such that it extends the interoperability to not only NS-2 but also C++Sim and the other sequential simulator. The lack of generality is the most disadvantageous aspect of the DEVS-NS2 comparing to the DEVS BUS framework. But, the DEVS-NS2 has connections among DEVS models and NS-2 models through a one-to-one mapping method which means that one DEVS object is mapping with one NS-2 object. A DEVS model triggers its matching NS-2 model directly. As a result, the DEVS-NS2 doesn't cause a bottleneck of message passing message passing among DEVS models and NS-2 models. On the contrary,

every message is processed in one component, a centralized deliverer, to be delivered to destinations in DEVS BUS framework. This message passing mechanism results a bottleneck. Consequently, we find the similarities between the DEVS-NS2 and the DEVS BUS framework. First of all, both the DEVS-NS2 and the DEVS BUS framework are centralized coupling. Scheduling approaches are very similar, too. As we presented earlier in this chapter, the DEVS-NS2 has the NS-2 Event Queue Agent model that connects to the NS-2's event queue through the DEVS NS-2 Interface. The NS-2 Event Queue model controls the NS-2's event queue by triggering NS-2's functions to handle the event queue. The objective of this model is to synchronize DEVS and NS-2. Similar to the NS-2 Event Queue Agent model, the DEVS BUS framework contains one scheduler that handles every message among models for time synchronization. Table 3 shows the comparison between the DEVS-NS2 and the DEVS BUS framework.

	DEVS-NS2	DEVS BUS framework
Bottleneck	Very low	Higher than DEVS-NS2
Generality	Not good(only with NS-2)	Good
Synchronization	Good	Good
Distribution simulation	No	No
Reusability	Good	Good
Developing cost	Good	Good

Table 3. Comparison between DEVS-NS2 and DEVS BUS framework

The DEVS-NS2 has one advantage and one disadvantage. Increasing the generality in order to integrate DEVS with not only NS-2 but also several different simulators is worthy of further research.

Although the approaches and implementation issues are different between the DEVS-NS2 and the DEVS BUS framework, the goals of increasing modeling power and reducing modeling development cost are the same.

### ***3.5. Comparison between DEVS-NS2 and OPNET***

OPNET is a widely used commercial product and its availability is strictly limited. The simulator is available with full source code for all network component modules, but the code for the simulation engine is not supplied. Although there exist examples and exhaustive documentation, additional support, available through OPNET mailing list, requires additional maintenance license. Also, there are a number of additional OPNET packages that require separate purchase. The main programming language in OPNET is C (recent releases support C++ development). The initial configuration such as network topology setup and parameter setting is usually achieved using a Graphical User Interface (GUI), a set of XML files or through C library calls. Simulation scenarios (e.g., parameter change after some time, topology update, etc.) usually require writing C or C++ code; although in simpler cases one can use special “scenario” parameters (e.g., link fail/restore time). OPNET modules implementations is very complex in contrast to NS-2’s relatively well designed C++ objects. OPNET’s

implementations are monolithic and use a number of global variables. Modifications of OPNET behavior are difficult, error-prone and require significant effort and amount of time. Since OPNET's network equipment models are very detailed, and in fact the simulation process closely reflects the processing that happens in real-world equipment, OPNET's simulation is relatively slow and requires powerful workstations. The GUI conveniently models network elements and protocols; this is really helpful for a user in understanding how the things work, but the lack of scripting language is limiting from a developer's point of view.

NS-2 seems to be completely free for both educational and commercial purposes although some older code explicitly grants rights to educational type of use only. The simulator is available with a full source code, validation tests, a rich set of examples and a good manual. Moreover, additional support may be provided by the ns-2 user's mailing list. NS-2 has been used in a great number of research projects in academia and is the most often used simulator in research projects related to IP networks. The main reasons behind this popularity are the fact that it is usually free and easy to use and that many well-recognized scientists have contributed significant amount of work to this project. Thus, probably most modern features related to IP networks are implemented in NS-2 and even if not merged with the distribution, they could be found somewhere in the Internet. Many researchers contribute to NS-2 thus continually updating the simulator with a reliable implementations for new protocols, queuing disciplines, etc.

In DEVS-NS2 environment, the details of low level network with protocol and component description is modeled by NS-2 while DEVS serves as controller by modeling

the high level behavior of target network models and interaction of the associated actors. The primary advantage of the DEVS-NS2 simulation comparing to OPNET simulation is that not only a network simulation but also a real environmental simulation are possible. On one hand, because OPNET simulation considers only packet transmitting related simulation such as network protocol development, transmitting delay, and so on, a receiving node just discards received packet in the simulation. On the other hand, in DEVS-NS2 simulation, a node doesn't discard packets that may include very important data. Once a node receives packets, it may use information for secondary reactions. As a result, the DEVS-NS2 is capable of doing what OPNET alone cannot do.

DEVS-NS2 environment reuses the network protocol libraries of NS-2's. The remarkable advantage of DEVS-NS2 compared to OPNET is usually free for the NS-2's network libraries and an additional support. The NS-2's libraries appear to be as good as the libraries of OPNET since, as mentioned earlier, they are updated with a reliable implementation for new protocols by many researchers in academia. Were the source codes of OPNET simulation engine to be available, an integration with OPNET would be promising extension. (Another approach to integrate with OPNET might be to use middleware such as HLA. DEVS may be possible to be integrated with OPNET if OPNET opens channels to be connected to HLA.) The above reasons show the advantages of DEVS-NS2 environment.



Table 4 shows the comparison of DEVS-NS2 and OPNET.

	OPNET	DEVS-NS2
Cost	highly expensive commercial software	completely free, open-source software
Speed	quite slow	fast
Availability	available with source code for simulation modules (except for restricted protocols).	available with full source code, validation tests and examples.
Support	<ul style="list-style-type: none"> <li>- excellent manual</li> <li>- mailing list (maintenance license required)</li> <li>- source code and examples</li> </ul>	<ul style="list-style-type: none"> <li>- good manual</li> <li>- publicly available mailing list</li> <li>- source code and examples</li> </ul>
Scenario	<ul style="list-style-type: none"> <li>- GUI, XML, imports</li> <li>- scenario parameters</li> <li>- Static (pre-defined)</li> </ul>	<ul style="list-style-type: none"> <li>- OTcl scripts (or C++)</li> <li>- DEVS modeling</li> <li>- Dynamic</li> </ul>
Components	<ul style="list-style-type: none"> <li>- C/C++</li> </ul>	<ul style="list-style-type: none"> <li>- DEVS (higher level)</li> <li>- C++ NS-2 libraries (lower level)</li> </ul>
Summary	<ul style="list-style-type: none"> <li>- widely used in military projects</li> <li>- run on powerful workstations</li> </ul>	<ul style="list-style-type: none"> <li>- popular in research projects in academia</li> <li>- well-defined mathematical formalism specification</li> </ul>

Table 4. Comparison of DEVS-NS2 and OPNET

## CHAPTER 4. EXPERIMENT AND RESULTS

In the chapter 3, the integration of DEVS and NS-2 is explained. This chapter shows an example that only the DEVS-NS2 can simulate, but an NS-2 simulation alone cannot. The primary advantage of the DEVS-NS2 simulation compared with an NS-2 simulation is that not only a network simulation but also a real environmental simulation are possible. On one hand, because the NS-2 simulation considers only packet transmitting related things such as network protocol development, transmitting delay, and so on, a receiving node just discards received packets in the simulation. On the other hand, in a DEVS-NS2 simulation, a node doesn't discard packets that may include very important data. Once a node receives packets, it may use information for secondary reactions. The simulation scenario and its network topology are presented first, and the DEVS atomic models, which are used in this example, are introduced in the following section. Lastly, the simulation results are analyzed in the last part of this chapter.

### *4.1. Scenario*

The purpose of this scenario is to propose the behavior of a sensor node in the wireless sensor networks. The basic idea is that sensor nodes send packets while they don't detect anything. The packets are named as "alive packets". Sending alive packets requires energy but, it may help for effective decision making. Through the simulation, the effectiveness of alive packets is proven in regard to decision making. The Figure 26 shows the schematic of this scenario.

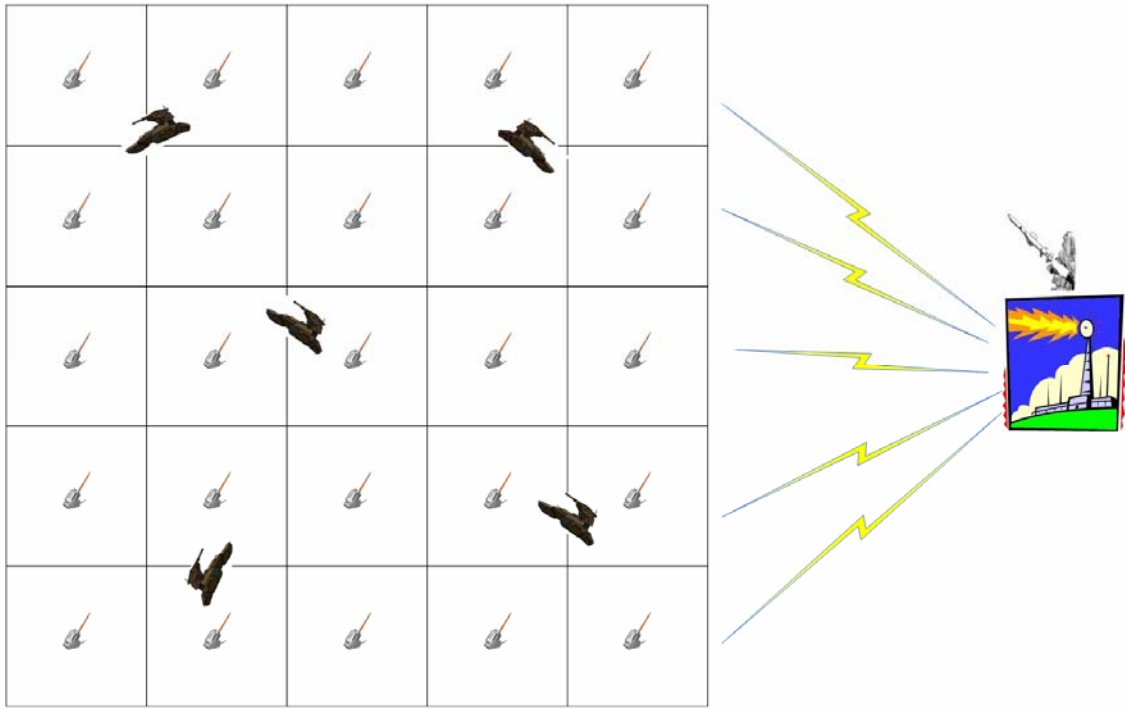


Figure 26. Schematic of the scenario

There are twenty five sensor nodes in a battle field and one base station. All the sensor nodes have their own sensing radius and the sensor nodes communicate with the base station through a radio based wireless channel. The base station has missiles and is ready to launch to attack enemy tanks. Five tanks are also in the battle field and are moving. Once sensor nodes detect tanks, they send detecting packets to the base station constantly while the tanks are in the sensor nodes' sensing area. At the same time, sensor nodes which detect nothing in their sensing area send alive packets to the base station with an alive packet interval time. If the base station node receives packets, then

it distinguishes packets as detecting packets or alive packets. If the base station decides the packet is a detecting packet, then the base station launches missiles to the point of a sensor node that is the source node of the detecting packet. Sometime later after the base station launches the missile, the missile arrives at the destination and explodes objects that are within the missile's explosive area. The tanks are to keep moving until they are destroyed by a missiles attack, and the sensor nodes are also to keep working on detecting tanks and sending packets that are either detecting packets or alive packets until either they are destroyed by missiles or they run out of energy. In this simulation, the time which all the sensor nodes are destroyed or all the tanks are destroyed is measured. If the base station doesn't receive any packets during an alive packet interval time, the base station considers that all the sensor nodes are destroyed and there is no way to detect enemy tanks. As soon as all the sensor nodes are destroyed, the base station gets to know that, and some action is needed to defend from an attack. The behavior of sensor node's sending alive packets is expected to be very effective for a defending system. In the next section, the DEVS models which are needed for this simulation are presented.

#### ***4.2. DEVS Model Behavior***

The schematic architecture of the modeling is shown in Figure 27. We may divide the schematic architecture into 3 parts, the DEVS modeling, the NS-2 modeling, and the DEVS and NS-2 interface.

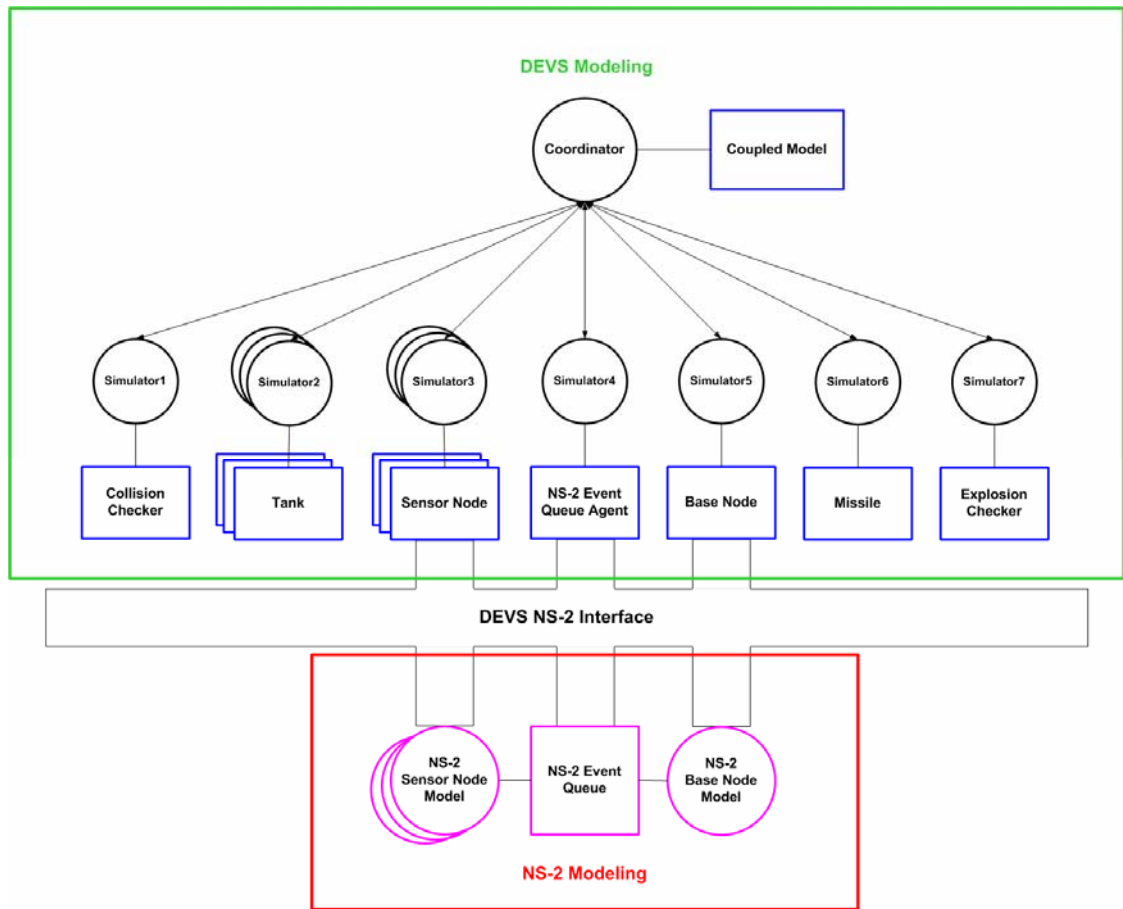


Figure 27. Schematic Architecture of the Modeling

There are seven DEVS atomic models which are the sensor node model, the base node model, the tank model, the missile model, the collision checker model, the explosion checker model, and the NS-2 Event Queue Agent model. The sensor node model and the base node model have connections with their matching NS-2 node models. The sensor node model, the tank model, the collision checker model, and the NS-2 Event Queue Agent model are based on the previously explained models in the chapter 3, but the previously explained models are so simple that we upgrade these models to more

complex and models adjust to this example. The other three models that are the base node model, the missile model, and the explosion checker model are newly developed. The DEVS atomic model's behavior is introduced in details in the following sections.

#### 4.2.1. The Sensor Node Model

The sensor node model is based on the model which is introduced in section 3.2.1. But, we modify the model by adding and removing certain behaviors of the sensor nodes. The biggest change is adding the “send alive packet” state to it. This is our new approach to define the sensor node's behavior in wireless sensor networks. We expect that sending alive packets would increase the decision making rate. So, the state diagram looks like Figure 28.

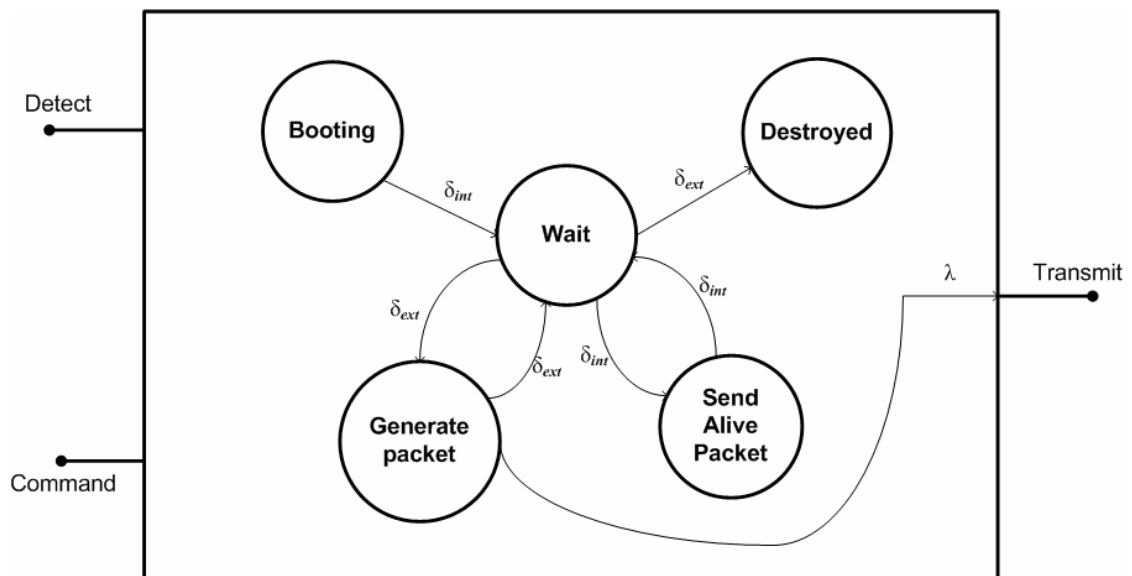


Figure 28. Sensor Node Model State Diagram

When the simulation starts, the sensor nodes are initiated with the “Booting” state. Once the sensor nodes are booted, they achieve the “Wait” state and wait for external inputs in regards to either detecting moving object events or moving object disappear events. The behavior of detecting moving objects and generating packets are as same as we described in the previous example in section 3.2.1. Unless the sensor nodes receive any types of external outputs in the “Wait” state during the time interval of alive packets, they enter into “Send Alive Packet” state and become “Wait” state again after sending one alive packet. The sensor nodes achieve the “Destroyed” state if they get exploded by missiles or they run out of energy. After the sensor nodes achieve the “Destroyed” state, they won’t attain the “Wait” state again. The following DEVS formalism represents the sensor node’s behavior that is explained above.

$$X = \{ \text{detect, command} \}$$

$$Y = \{ \text{transmit} \}$$

$$S = \{ \text{booting, wait, generate\_pkt, send\_alive\_pkt, destroyed} \}$$

$$\delta_{\text{int}} : \text{booting} \rightarrow \text{wait}$$

$$\text{wait} \rightarrow \text{send\_alive\_pkt}$$

$$\text{send\_alive\_pkt} \rightarrow \text{wait}$$

$$\delta_{\text{ext}} : \text{wait} * \text{detect} \rightarrow \text{generate packet}$$

$$\text{generate packet} * \text{detect} \rightarrow \text{wait}$$

wait \* command  $\rightarrow$  destroyed  
 $\lambda$  : generate packet  $\rightarrow$  transmit  
 $\tau_a$  :  $\infty$  if s = generate packet, destroyed  
           time interval for alive packets if s = wait  
           0.001 if s = send\_alive\_pkt  
           Randomly generate number(0~1) if s = booting

#### 4.2.2. The Base Node Model

The base node model has not been introduced in the previous example. The previous example doesn't react according to received packets because that concentrates on only integration issues. However, the purpose of this example is to show that the DEVS-NS2 is capable of doing what the NS-2 alone cannot do. That's why the base node model is newly introduced in this section. Prior to explaining the behavior of the base node model, it needs to be mentioned that the DEVS base node model connects directly to the NS-2 base node model the same way the sensor node model does. Figure 29 presents the state diagram of the DEVS base node model.



The base node starts with the “Wait” state initially. The base node model receives packets from its mapping NS-2 node. If the packets are alive packets, then the base node becomes the “Receive Alive Packet” state and recognizes that the sensor nodes which generate the packets are still alive. In turn, the base node discards the packets. While, if the packets are detecting packets, the base node enters the “Receive Detecting Packet” state and becomes the “Fire Missile” state consequently. In the “Fire Missile” state, an output function ( $\lambda$ ) is called and the base node sends a message to the output port. The message is to launch missiles. The output port connects with the input port of the missile model that is discussed in detail later. The base node’s DEVS formalism is presented below.

$$X = \{ \text{receive\_packet} \}$$

$$Y = \{ \text{command} \}$$

$$S = \{ \text{wait, receive\_detecting\_pkt, receive\_alive\_pkt, fire\_missile} \}$$

$$\delta_{\text{int}} : \text{receive\_alive\_packet} \rightarrow \text{wait}$$

$$\text{receive\_detecting\_pkt} \rightarrow \text{fire\_missile}$$

$$\text{fire\_missile} \rightarrow \text{wait}$$

$$\delta_{\text{ext}} : \text{wait} * \text{receive\_pkt} \rightarrow \text{receive\_alive\_pkt}$$

$$\text{wait} * \text{receive\_pkt} \rightarrow \text{receive\_detecting\_pkt}$$

$$\lambda : \text{fire\_missile} \rightarrow \text{command}$$

$$ta : \infty \text{ if } s = \text{wait}$$

$$0.001 \text{ if } s = \text{receive\_alive\_pkt, receive\_detecting\_pkt, or fire\_missile}$$

#### 4.2.3. The Tank Model

The tank model inherits and reuses the moving object model that was introduced in the section 3.2.2. The difference is that the tank model moves following randomly generated routes during a simulation rather than pre-determined routes. The initial positions are also determined randomly as soon as the simulation starts. The rest of the other behaviors are exactly the same as the parental moving object model.

#### 4.2.4. The Missile Model

Once the base node receives packets which are generated because of detecting

tanks illustrated in section 4.2.2, the base node needs to launch missiles for the purpose of destroying tanks. This section presents the missile model. Figure 30 shows the state diagram of the missile model's behavior.

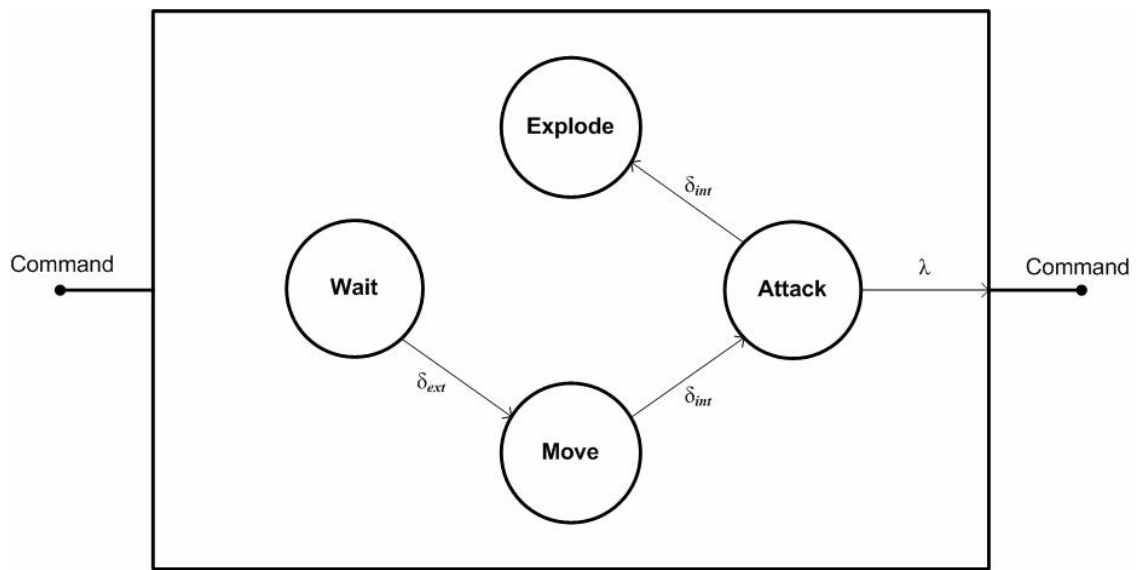


Figure 30. Missile Model State Diagram

There exist 4 states: “Wait”, “Move”, “Attack”, and “Explode”. The initial state is “Wait” and transits to the “Move” state if the model receives messages from the input port. The missile heads to the targets whose positions are included in the received messages. It is assumed that missiles can reach any position from the base node in 5 seconds. The internal state transition function ( $\delta_{int}$ ) is called to change the missile’s state from “Move” to “Attack” after the “Move” state’s time advance of 5 seconds. The missile model generates the output messages and the messages are sent to the explosion checker model that is discussed later. It is also assume that the missiles require very

short times(0.001 seconds) to attack a target. Finally, the missile model becomes the “Explode” state. The missile model’s DEVS formalism is written as follow:

$$X = \{ \text{command} \}$$

$$Y = \{ \text{command} \}$$

$$S = \{ \text{wait, move, attack, explode} \}$$

$$\delta_{\text{int}} : \text{move} \rightarrow \text{attack}$$

$$\text{attack} \rightarrow \text{explode}$$

$$\delta_{\text{ext}} : \text{wait} * \text{command} \rightarrow \text{move}$$

$$\lambda : \text{attack} \rightarrow \text{command}$$

$$ta : \begin{cases} \infty & \text{if } s = \text{wait, explode} \\ 0.001 & \text{if } s = \text{attack} \\ 5 & \text{if } s = \text{move} \end{cases}$$

#### 4.2.5. The Collision Checker Model

Like the tank model, the collision checker model that is seen in the section 3.3.3 is reused in this scenario. But, we modify the checking method according to this example. The original model checks the collision by measuring distances between each moving object and all the sensor nodes. However, the original collision checking

method may decrease the computational power as the number of the sensor nodes increases since the model checks every sensor nodes with each moving object. In order to prevent the computational speed down, the collision checker model examines several sensor nodes which are close to each tank. Figure 44 shows our way to check the collision.

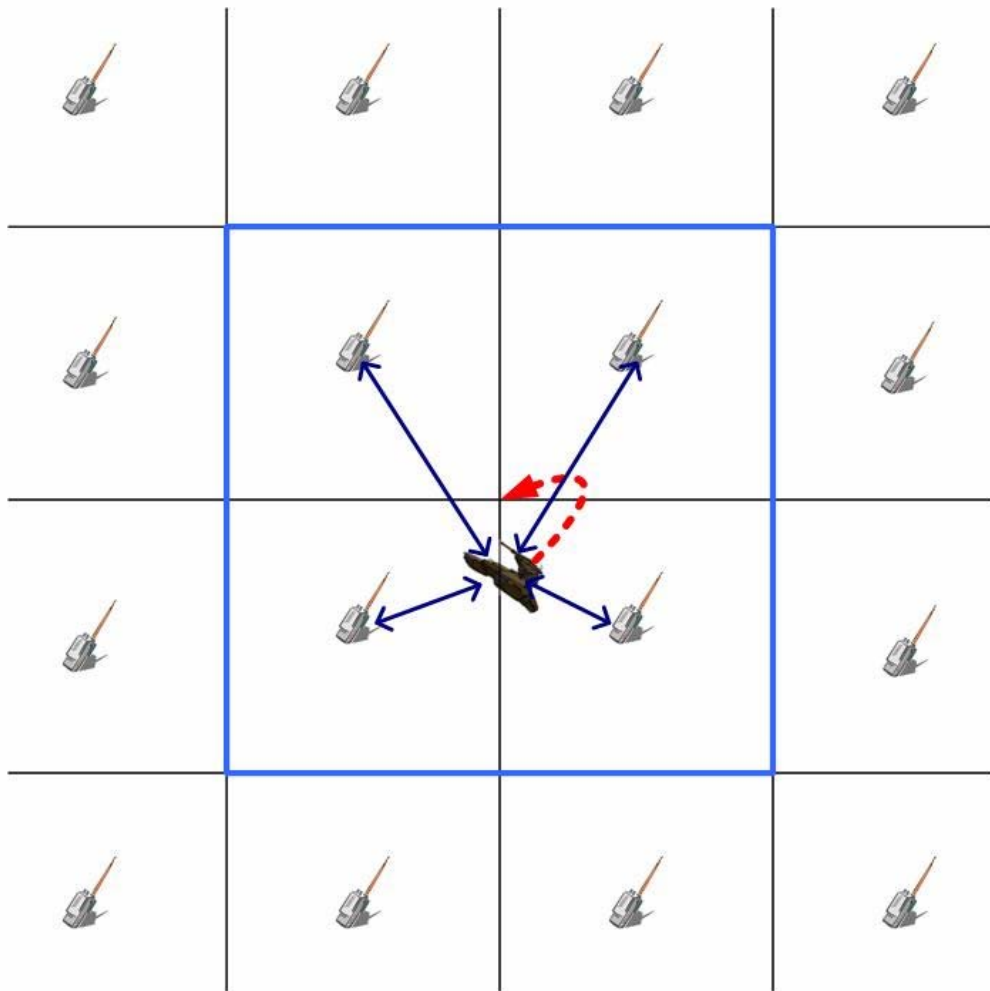


Figure 31. Collision Checking Method

In order to pick sensor nodes that are checked with one tank, a virtual position of a tank is decided by finding the closest position at which four sensor nodes meet from a tank's current position. Once four nodes are picked, the distances between a tank and a sensor node is measured. The sensor node which has the shortest distance to the tank is the node that is able to detect the tank. Ideally, we may expect  $O\left(\frac{\text{number of sensor nodes}}{4}\right)$  speed up comparing to the original method that checks all sensor nodes.

#### 4.2.6. The Explosion Checker Model

The explosion checker model works like the collision checker model but, it is triggered when it receives messages through the input port while the collision checker model is called by phi function. The main objective of this model is to find sensor nodes and tanks destroyed by a missile attack. The state diagram of the explosion checker model is explained in Figure 32.

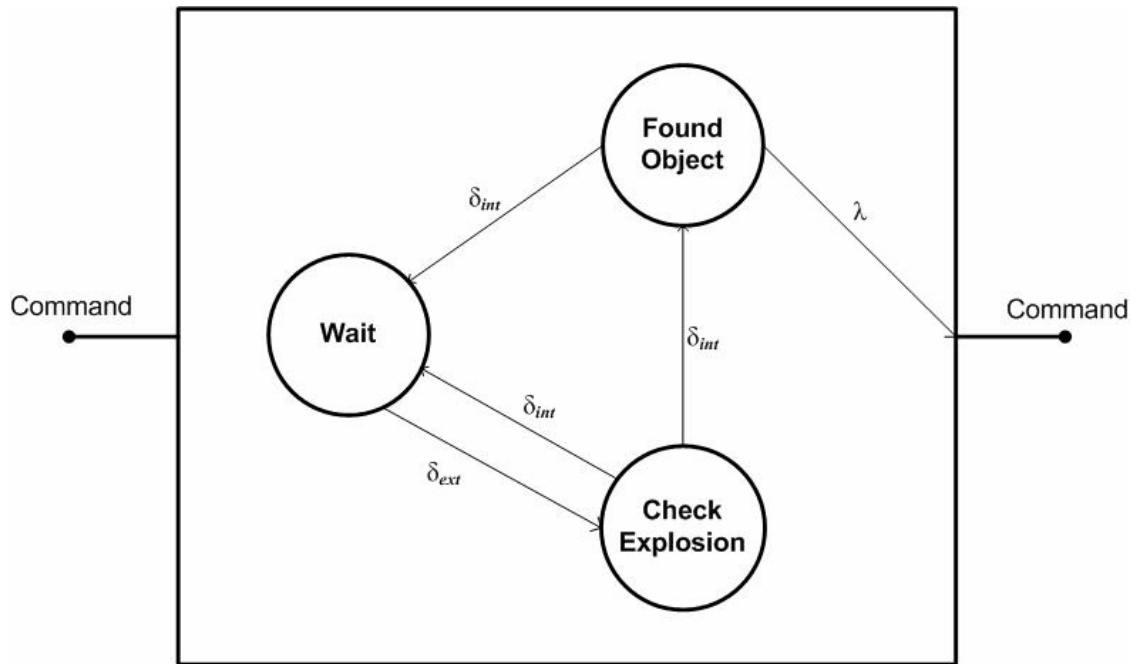


Figure 32. Explosion Checker Model State Diagram

As soon as the simulation starts, the explosion checker model is the “Wait” state with  $\infty$  of time advance. If the model receives messages at the input port, “command”, the state becomes the “Check Explosion” state. The messages may come from the missile models when the missiles are exploded. The explosion checker model examines the destruction of sensor nodes and tanks in regard to the exploded missiles’ position. If there exists at least one sensor node or tank, it enters the “Found Object” state and becomes the “Wait” state after sending messages to the models which are destroyed by the missile attack. While, if the explosion checker model can’t find anything around the missiles’ explosion area, it goes back to the “Wait” state. The DEVS formalism of the explosion checker model is depicted below:

$$X = \{ \text{command} \}$$

$$Y = \{ \text{command} \}$$

$$S = \{ \text{wait}, \text{check\_explosion}, \text{found\_object} \}$$

$$\delta_{\text{int}} : \text{check\_collision} \rightarrow \text{found\_object}$$

$$\text{check\_collision} \rightarrow \text{wait}$$

$$\text{found\_object} \rightarrow \text{wait}$$

$$\delta_{\text{ext}} : \text{wait} * \text{command} \rightarrow \text{check\_collision}$$

$$\lambda : \text{found\_object} \rightarrow \text{command}$$

$$\text{ta} : \infty \text{ if } s = \text{wait}$$

$$0.0 \text{ if } s = \text{check\_explosion}, \text{found\_object}$$

#### 4.2.7. The NS-2 Event Queue Agent Model

Since the DEVS-NS2 is proved in term of the successful integration in the previous chapter, the NS-2 Event Queue Agent model is reused.

### 4.3. Simulation Results

Under the circumstances of twenty five sensor nodes, five tanks, and one base station, two different sensor node's behaviors are simulated. The first simulation is

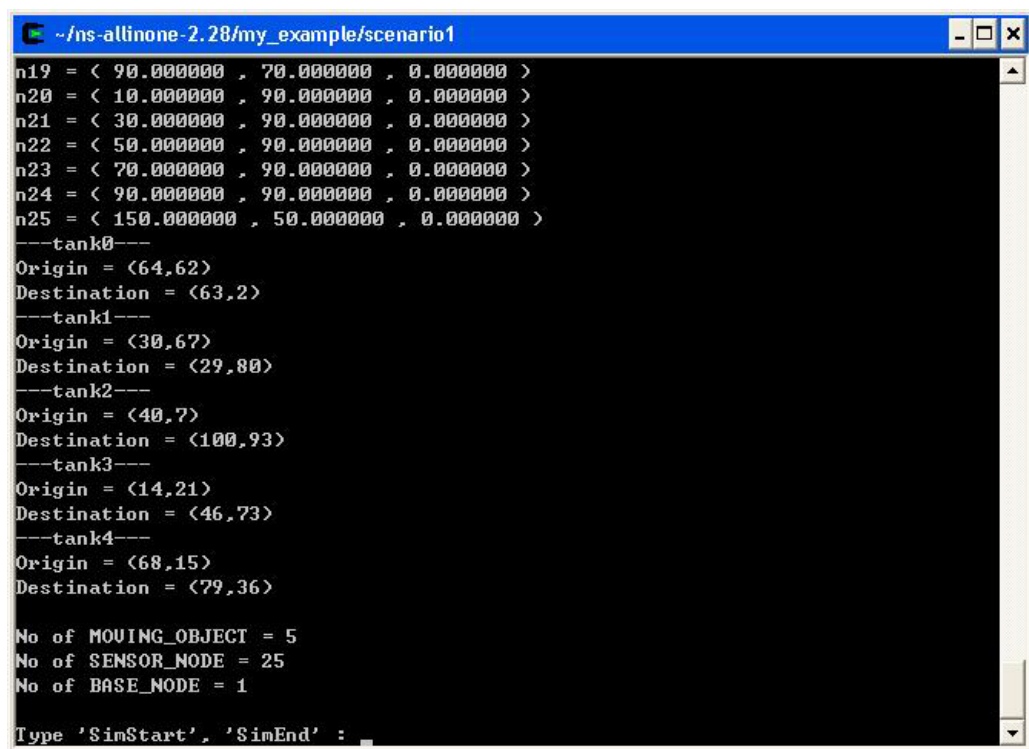


done with conventional sensor node's behavior, and the second simulation is run with our proposed sensor node's behavior including sending alive packets. Excluding the difference of sensor node's behaviors, all the simulation variables are in the same condition. The energy consumption, the number of tanks destroyed, and the number of sensor nodes destroyed are measured during 1500 simulation time. If all the tanks are destroyed, the simulation needs to be stopped. However, unfortunately, our simulation can't destroy all the tanks for 1500 seconds of simulation time. The assumptions for the simulation are follows:

- All the sensor nodes and tanks are in the 100\*100 simulation space.
- The sensor node sends alive packets with a 1 second simulation time interval.
- The sensor node covers one square size(20\*20) of sensing area.
- The initial energy is set to 100 joules in all sensor nodes.
- It takes 5 seconds of simulation time for the missile to arrive at the destination position from the base station node.
- The missile's destructive radius is 7.
- The probability of destruction of both the sensor nodes and the tanks is 50% if the objects are in the missile's destructive area.
- The tanks are initially positioned arbitrarily and move on a randomly generated route before they get destroyed.

The nodes are configured the same as the configuration which is presented in

section 3.3.1. Figure 33 depicts the simulation console that shows the initialization of the sensor nodes, tanks, and the base station node; and is ready to run. The nodes' positions are set and the original position and the first destination positions of the tanks are decided.



```

~\ns-allinone-2.28\my_example\scenario1
n19 = < 90.000000 , 70.000000 , 0.000000 >
n20 = < 10.000000 , 90.000000 , 0.000000 >
n21 = < 30.000000 , 90.000000 , 0.000000 >
n22 = < 50.000000 , 90.000000 , 0.000000 >
n23 = < 70.000000 , 90.000000 , 0.000000 >
n24 = < 90.000000 , 90.000000 , 0.000000 >
n25 = < 150.000000 , 50.000000 , 0.000000 >
---tank0---
Origin = <64,62>
Destination = <63,2>
---tank1---
Origin = <30,67>
Destination = <29,80>
---tank2---
Origin = <40,7>
Destination = <100,93>
---tank3---
Origin = <14,21>
Destination = <46,73>
---tank4---
Origin = <68,15>
Destination = <79,36>

No of MOVING_OBJECT = 5
No of SENSOR_NODE = 25
No of BASE_NODE = 1

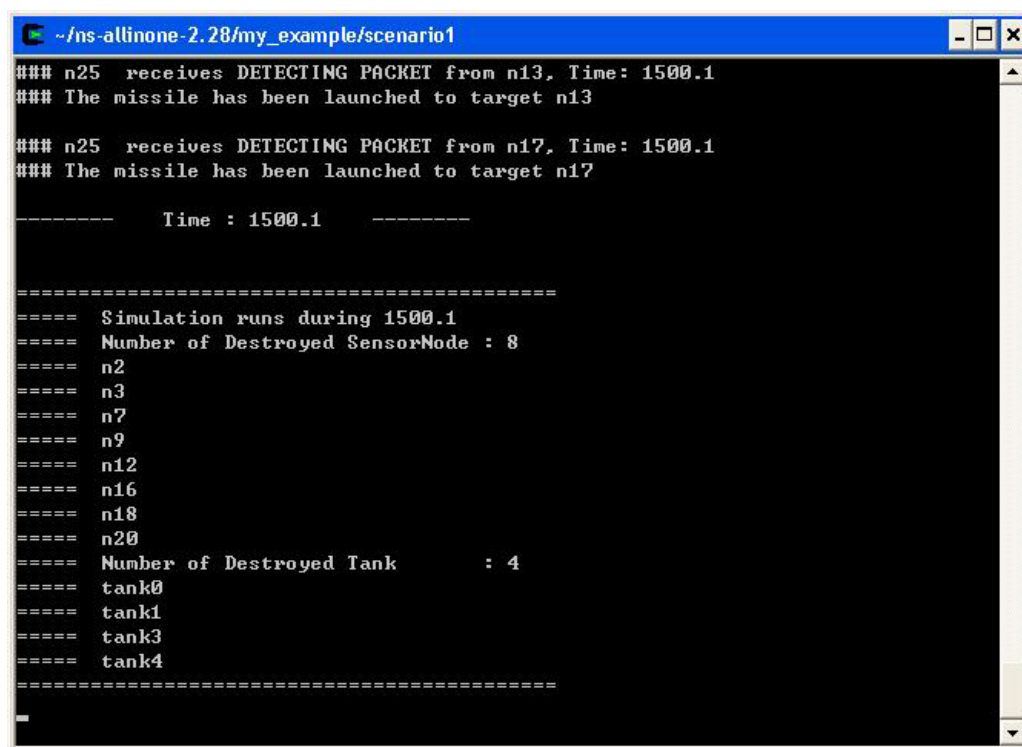
Type 'SimStart', 'SimEnd' :

```

Figure 33. Simulation Initialization

The following two figures: Figure 34 and Figure 35, are the simulation consoles when the simulations are stopped. The first simulation uses the conventional sensor node model that has no behavior of sending alive packets, but the sensor node model with sending alive packet behavior is used in the second simulation. The result of the first

simulation is shown in Figure 34. Figure 35 is the result of the second simulation. The numbers of destroyed sensor nodes in both simulations are the same and the destroyed tanks are also the same in both simulations. The same results in both simulations prove that alive packets don't affect the base station's behavior in regard to launching missiles. During 1,500 seconds of simulation time, eight sensor nodes and four tanks were destroyed.



```

~/.ns-allinone-2.28/my_example/scenario1
### n25 receives DETECTING PACKET from n13, Time: 1500.1
### The missile has been launched to target n13

### n25 receives DETECTING PACKET from n17, Time: 1500.1
### The missile has been launched to target n17

-----      Time : 1500.1      -----

=====
====  Simulation runs during 1500.1
====  Number of Destroyed SensorNode : 8
====  n2
====  n3
====  n7
====  n9
====  n12
====  n16
====  n18
====  n20
====  Number of Destroyed Tank      : 4
====  tank0
====  tank1
====  tank3
====  tank4
=====

```

Figure 34. Simulation Console with no alive packet sending

```

- /ns-allinone-2.28/my_example/scenario1
### n25 receives DETECTING PACKET from n11, Time: 1500.1
### The missile has been launched to target n11

### n25 receives DETECTING PACKET from n8, Time: 1500.1
### The missile has been launched to target n8

----- Time : 1500.1 -----

=====
==== Simulation runs during 1500.1
==== Number of Destroyed SensorNode : 8
==== n2
==== n3
==== n7
==== n9
==== n12
==== n16
==== n18
==== n20
==== Number of Destroyed Tank : 4
==== tank0
==== tank1
==== tank3
==== tank4
=====

```

Figure 35. Simulation Console with 1 second alive packet interval

In addition to determining the number of sensor nodes and tanks destroyed during the simulation, the energy consumption required for networking activities is also measured. Since the assumption shows that alive packets are generated every second during 1,500 seconds, the sensor nodes, which have sending alive packet behavior, generate and send about 1,500 packets more than the conventional sensor nodes do. As a result, the sensor nodes in the second simulation consume more energy compared to the sensor nodes in the first simulation according to the number of alive packet sent. Figure 36 is the chart of the total number of packets generated in both the first and second simulation.

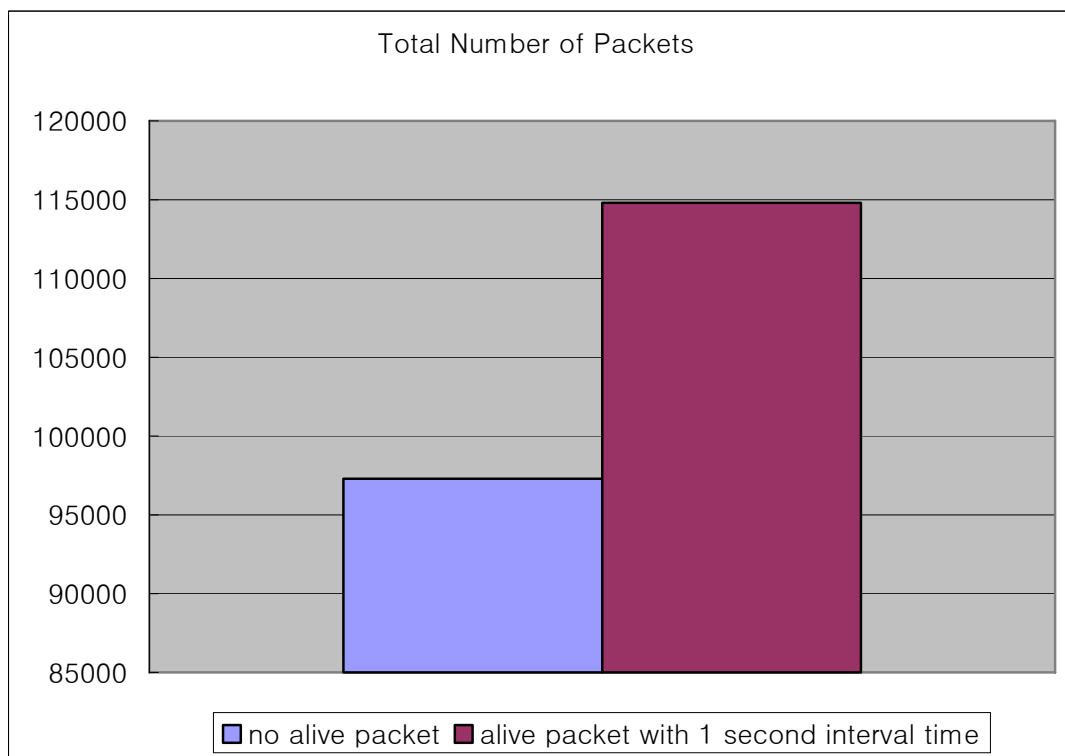


Figure 36. Total number of packets

The energy consumed in each sensor node for both simulations are demonstrated in Figure 37. All the sensor nodes which send alive packets consume more energy than the others. The energy consumption of the eight destroyed sensor nodes 2, 3, 7, 9, 12, 16, 18, and 21 are not measured. Figure 38 shows the total energy consumption in both simulations.

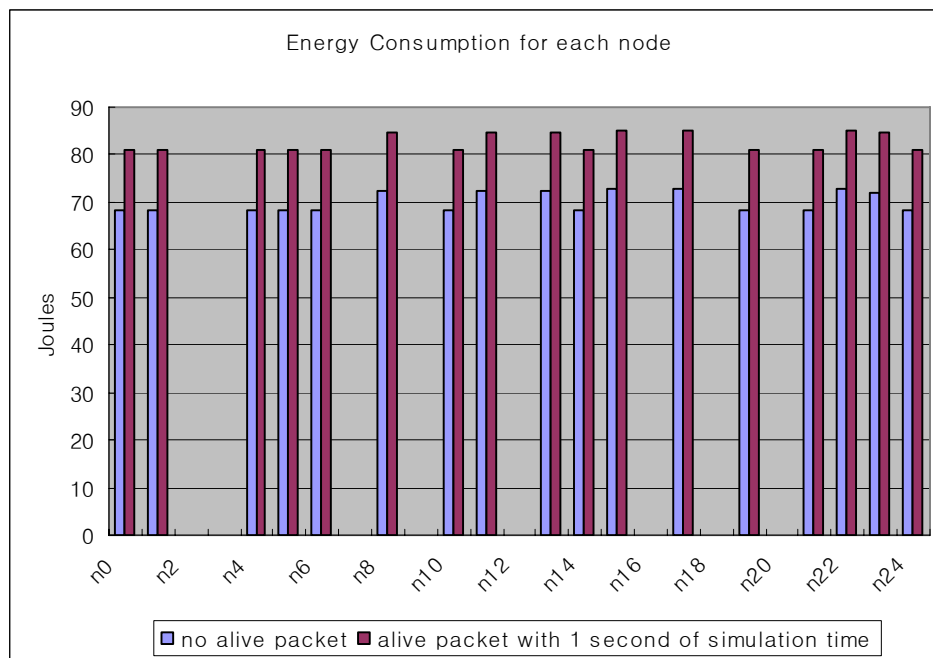


Figure 37. The energy consumption for each sensor node

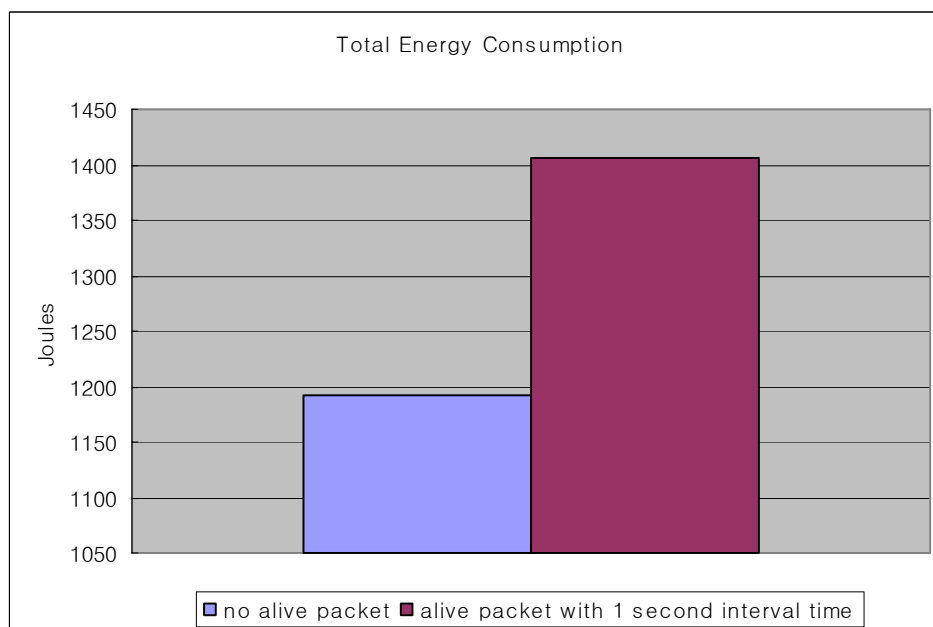


Figure 38. The total energy consumption

Unfortunately, the simulation is stopped before any sensor node uses up its energy. But, the base station gets to know whether sensor nodes are deactivated because of the lack of energy after the alive packet interval time if the base station doesn't receive alive packets from those nodes. It helps for the base station to make a reaction for deactivated sensor nodes. Through this simulation, it is recognized that the behavior of sending alive packets is disadvantageous to the energy consumption aspect, but the behavior is advantageous to make a decision.

## CHAPTER 5. CONCLUSION AND FUTURE WORKS

### 5.1. Conclusion

The main objective of this research is to design and implement an interoperable simulation with DEVS and NS-2. DEVS and NS-2 models can participate in a simulation, and their appropriate role portions are decided in order to increase model reusability and reduce the model development cost. To accomplish these goals, the time synchronization between two simulators and the aggregating method to match pairs of DEVS and NS-2 models are required to be performed mainly. Our approach to synchronize both DEVS and NS-2 scheduling methods is forming one DEVS model which includes the event queue of NS-2 and handles inserting and removing activities. As a result, the NS-2 process looks like the DEVS simulator's behavior. In addition, DEVS models and their matching NS-2 models are connected with each other by applying a one-to-one mapping system. Moreover, because the DEVS-NS2 is not a centralized system, the speed degradation caused by message bottlenecks is not expected.

To prove the concrete integration of the DEVS-NS2, a very simple network topology model with seven sensor nodes, one tank, and one base node is built. The number of generated packets and the energy consumption for each sensor node in both the DEVS-NS2 simulation and the NS-2 alone simulation are measured. The exact same results in two simulations conclude the success of the integration.

In the second example, the new behavior of sensor nodes in wireless sensor networks is proposed. Generating and sending alive packets with some interval time



while sensor nodes don't detect any kind of moving object is a method to inform a base station of sensor nodes' aliveness. These activities require more energy consumption rather than a behavior of traditional sensor nodes. In the simulation, the new proposed behavior in term of alive packets consumes 17.96% more energy than the other behavior. Although energy saving is a big issue in wireless sensor networks, finding enemies and defending from their attacks are more important from a defending system's point of view.

## ***5.2. Future works***

There are a few issues which are worthy of further research. The issues include refining models and finding optimal solutions, generalizing the interface with DEVS and other simulators, and expanding to a distributed simulation. Refining models allows simulations to be more reasonable. The behaviors of models which are designed in this thesis are not realistic but virtually assumed. If the real battle field data such as behaviors of tanks, missiles, and sensor nodes can be applied to the models, a simulation increases its realism. In this thesis, the new behavior of sensor nodes in wireless sensor networks is proposed, and the control variable of the interval time for sending alive packets is set as 1 second. However, finding the optimal interval time in regard to the severity of a battle is a challenging research area for the tradeoff between energy consumption and effective decision making.

The second issue is the generalization of the interface with DEVS and other simulations. The DEVS-NS2 is integrated with DEVS and NS-2 only. The integration

with other network simulators such as OPNET and C++Sim increases modeling power and reduces model development cost. The third issue is to validate the concreteness of the DEVS-NS2 environment. In chapter 3 we measured the energy consumption and the total number of packets generated in both the DEVS-NS2 simulation and the stand-alone NS-2 simulation, and concluded that the DEVS is integrated with NS-2 is valid because the simulation results are exactly same. However, we need to model and simulate the several examples more to get more confidence in validity of the DEVS-NS2 environment. For future work, we intend to provide a proof of correctness of the implementation using the DEVS Abstract Simulator concepts. An example of this approach appears in [21].

Lastly, expanding to a distributed simulation is a promising area for further research. The sequentially processed interoperable simulation may cause severe speed degradation as the size of the network increases. To get more faster results in a large scale network topology simulation environment, a distributed simulation may provide a simulation, e.g., [7, 13].

Through further research on those issues, it is expected that the DEVS-NS2 will become a reliable simulator for general in network related simulations.

## REFERENCES

- [1] K. Fall and K. Varadhan, “*The ns Manual*,” <http://www.isi.edu/nsnam/ns> , 2005
- [2] B.P. Zeigler, T.G. Kim, and H. Praehofer, “Theory of Modeling and Simulation,” 2nd ed., New York: Academic Press, 2000.
- [3] OPNET Technologies, OPNET Modeler, Commercial, Information, <http://www.opnet.com/products/modeler/home.html>
- [4] Cygwin User’s Guide, <http://www.cygwin.com>
- [5] MH. Hwang, Open DEVS C++, <http://sourceforge.net/projects/odevspp>
- [6] Y.G. Kim, J.H. Kim, and T.G. Kim, “Heterogeneous Simulation Framework Using DEVS BUS,” *SIMULATION*, Vol. 79, Issue 1, January 2003
- [7] S. Park, “Cost-Based Partitioning For Distributed Simulation of Hierarchical Modular DEVS Models,” in *The Department of Electrical and Computer Engineering: Ph.D Dissertation*, the University of Arizona, 2003.

- [8] K.M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. 5, pp. 440-452, 1979.
- [9] R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, vol. 33, pp. 160-177, 1984
- [10] D.A. Jefferson and H. Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism," presented at The Society for Computer Simulation (SCS), La Jolla, California, 1985.
- [11] D.A. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404-425, 1985.
- [12] "DEVS-Java Reference Guide," <http://acims.arizona.edu>
- [13] B.P. Zeigler, Yoonkeon Moon, Doohwan Kim, and Jeonggeun Kim, "DEVS-C++: A High Performance Modeling and Simulation Environment," *29<sup>th</sup> Hawaii International Conference on System Sciences*, pp. 350, 1996
- [14] Y.K. Cho, B.P. Zeigler, H.J. Cho, H.S. Sarjoughian, and S. Sen, "Design Consideration for Distributed Real-time DEVS," *AI and Simulation Conference*,

Tucson, AZ, 2000

- [15] C. Shen, C. Srisathapornphat, and C. Jaikaeo, "Sensor Information Networking Architecture and Applications," *International Workshop on Pervasive Computing*, Toronto, Canada, August 2000
- [16] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," *IEEE Communication Magazine*, pp. 102-114, 2002
- [17] E. Cayirci, R. Govindan, T. Znati, and M. Srivastava, "Wireless Sensor Networks," *Computer Networks*, Volume 43, Issue 4, pp. 417-419, November 2003
- [18] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," *IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pp. 1567-1576, June 2002
- [19] T.V. Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," *ACM SenSys'03*, Los Angeles, California, November 2003
- [20] M. Little, C++SIM "User's Guide Public Release 1.5," <http://cxxsim.ncl.ac.uk>

- [21] J. Nutaro, "Time Management and Interoperability in Distributed Discrete Event Simulation," in *The Department of Electrical and Computer Engineering*: M.S. Thesis, the University of Arizona, 2003.