**DEVS model of an OmniEngine Control System**

**Department of Systems and Computer Engineering**

**Assignment 1 Report for SYSC 5104: Methodologies for Discrete-Event Modelling and Simulation**

**Fall 2011**

**Instructor: Prof. Gabriel Wainer**

October 30, 2011

*Author:*

*K. Waqas Muhi - 100629922*

**Original Proposal:**

**SYSC 5104 – Assignment 1 proposal**

Name: Kanwar Waqas Muhi  
Student ID: 100629922

WildPackets ([www.wildpackets.com](http://www.wildpackets.com)) is a company based in California specializing in network analysis tools such as packet sniffing software and others. The software of interest in this assignment would be their OmniPeek Network Analyzer; in particular the system which controls its main module – called the OmniEngine – which is responsible for all the analysis that takes place.

The assignment would look at the various functions the OmniEngine supports, how these work together, and how these can be modeled to give us more insight into the performance of the system. At the most basic level, the OmniEngine is responsible for “sniffing” (i.e. detecting) and collecting network packets via a specific network interface. The resulting “capture” consists of a packets collected from nearby devices of interest supporting various protocols such as WiFi and Ethernet.

The basic functions of the OmniEngine control system are as follows:

1. Connecting with the OmniEngine – the engine could be local or remote
2. Creating the specified capture on the engine.
3. Applying a filter if specified.
4. Starting the capture.
5. Stopping the capture.
6. Saving the resulting capture to a specified location.

The DEVS model would consist of various models representing the different states of the control system, the inputs/outputs, and the transitions involved.

**Description of the model actually designed:**

The top-level coupled model for the complete OmniEngine control system is shown below:

OmniEngine Control System

capture\_created

create\_capture

filter\_code

start\_capture

capture\_started

stop\_capture

channel\_num

capture\_stopped

The model was simplified, as compared to the one proposed, by taking out the “Connecting with the OmniEngine” and the “Saving the capture” functionalities. All the other functions were modelled as proposed.

The inputs represent the 3 different types of commands supported by the control system and are specified with integer values – 1 for create\_capture, 2 for start\_capture, and 3 for stop\_capture. A simplifying assumption made was that these commands would only be entered in the order shown in the figure i.e. create, start, stop.

The outputs inform the user of the command status i.e whether capture was created or not etc. The filter\_code gives the type of filter used (random number generated) and the channel\_num gives the channel number set (random integer).

The formal specifications of the OmniEngine Control System are given below:

In Ports: create\_capture, start\_capture, stop\_capture

Out Ports: capture\_created, filter\_code, capture\_started, channel\_num, capture\_stopped

OEControl = <X,Y,D,EIC,EOC,IC,SELECT>

X = {create\_capture, start\_capture, stop\_capture}

Y = {capture\_created, filter\_code, capture\_started, channel\_num, capture\_stopped}

D= {create\_module, start\_module, stop}

EIC = {(OEControl.create\_capture, create\_module.in),

(OEControl.start\_capture, start\_module.in),

(OEControl.stop\_capture, stop\_capture.cmd\_in)}

EOC = {(create\_module.capture\_created, OEControl.capture\_created),

(create\_module.filter\_code, OEControl.filter\_code),

(start\_module.capture\_started, OEControl.capture\_started),

(start\_module.channel\_num, OEControl.channel\_num),

(stop.capture\_stopped, OEControl.capture\_stopped)}

IC = {}

SELECT :{create\_module, start\_module, stop} = create\_module

Testing can be done by inputting one or more events at each of the 3 input ports in the order mentioned above and observing the output:

Input

00:00:10:00 create\_capture 1

00:00:30:00 start\_capture 2

00:00:50:00 stop\_capture 3

00:00:60:00 create\_capture 1

00:00:90:00 start\_capture 2

00:00:120:00 stop\_capture 3

Output

We expect to get 1 for all of capture specific outputs and random numbers representing the filter\_code and channel\_num. Sample output is shown below:

00:00:15:567 filter\_code 0.695611

00:00:17:482 capture\_created 1

00:00:36:435 channel\_num 9

00:00:40:756 capture\_started 1

00:00:55:699 capture\_stopped 1

00:01:10:202 filter\_code 7.47918

00:01:13:863 capture\_created 1

00:01:38:175 channel\_num 5

00:01:38:285 capture\_started 1

00:02:05:356 capture\_stopped 1

A description of the two coupled models and all the atomic models contained within is given starting from the next page.

The create\_module is a sub-model consisting of 2 components: a CreateCapture atomic model and a CreateFilter atomic model. Together, these simulate the creation of a capture with a specific filter (chosen randomly here). Both functions take a random amount of processing time to finish.

Create\_Module

Filter Code

Capture

Created

Filter Code

Capture

Created

Filter Done

Filter Go

Create\_Filter

Create\_Capture

Create Commandd

Create Command

Create\_module Formal specifications:

Create\_Module = <X,Y,D,EIC,EOC,IC,SELECT>

X = {Create Command}

Y = {Capture Created, Filter Code}

D={ Create\_Capture, Create\_Filter }

EIC = {( Create\_Module.Create Command, Create\_Capture.Create Command }

EOC = {( Create\_Capture.Capture Created, Create\_Module.Capture Created),

( Create\_Filter.Filter Code, Create\_Module.Filter Code)}

IC = {( Create\_Capture.Filter Go, Create\_Filter.Filter Go ) ,

(Create\_Filter.Filter Done, Create\_Capture.Filter Done)}

SELECT: Priority order (Descending) : Create\_Capture. Create\_Filter

Testing can be done by inputting one or more events at the input port – it should only respond to a 1 input evident from the timestamps of the outputs shown below – and verifying that a 1 is received for Capture Created and a random number for Filter Code.

00:00:10:00 in 2

00:00:50:00 in 1

00:00:60:00 in 3

00:00:120:00 in 1

00:00:150:00 in 1

Sample output is shown below.

00:00:55:567 filter\_code 8.92151

00:00:57:482 capture\_created 1

00:02:06:435 filter\_code 7.07847

00:02:10:756 capture\_created 1

00:02:41:009 filter\_code 8.83891

00:02:45:901 capture\_created 1

Following is a description of the two atomic models which make up this coupled model.

The Create\_Capture is an atomic model which takes in two inputs as shown in the figure above and generates two outputs. It continuously waits for a new command to arrive at the input port and sends off the single to Create\_Filter once a new Create Capture command is received. This sending off is simulated by generating a random processing time.

Upon receiving the Filter Done input, it proceeds to create the rest of the capture which is also simulated by generating a random processing time. It then sends the final Capture Created output to the output port.

Formal specifications for Create\_Capture = <X, Y, S, δint, δext, λ, ta>

X = {Create Command | Represented by integer value of 1, Filter Done}

Y = {Capture Created , Filter Go}

S = { Phase, sigma, wait\_for\_cmd, create\_filter, process\_cmd}

δext (S,e,X) {

Case port

Create\_Command:

Case Phase

Passive:

Check if wait\_for\_cmd = 1 and Command value is 1

Wait\_for\_cmd = 0 // set it to False

Create\_filter = 1 // set it to True

Sigma = generate random Processing Time;

Phase = busy;

busy:

// Ignore the request;

Filter\_Done:

Case Phase

Passive:

Check if create\_filter = 1

Create\_filter = 0 // set it to False

Process\_cmd = 1 // set it to True

Sigma = generate random Processing Time;

Phase = busy;

busy:

// Ignore the request;

}

λ (S) {

case phase

busy:

If Create\_filter = 1

//filter needs to be created, send the signal

Send (Filter\_go, 1);

If Process\_cmd = 1

// Capture created

Send (Capture\_created, 1);

Passive: // Should not happen.

}

δint (S) {

case phase

If Process\_cmd = 1

Process\_cmd = 0 // set it to False

Wait\_for\_cmd = 1 // Wait for new command

busy:

phase = passive;

sigma = infinity;

passive: //never happens

}

Create\_Capture was not tested individually, however, testing can be done by inputting one or more events – for each of the two input ports – and observing the outputs.

00:00:10:00 create\_command 1

00:00:50:00 filter\_done 1

00:00:60:00 create\_command 1

00:00:120:00 filter\_done 1

Create\_Filter simulates the creation of a filter by first generating a random number for the filter type then waiting for a random processing time. It outputs a done signal and the filter code.

Formal specifications for Create\_Filter = <X, Y, S, δint, δext, λ, ta>

X = {Filter Go| Represented by a value of 1}

Y = {Filter Done , Filter Code}

S = { Phase, sigma, start\_create, finish\_create}

δext (S,e,X) {

Case port

Filter\_Go:

Case Phase

Passive:

Check if start\_create = 1 and Filter\_Go value is 1

Start\_create = 0 // set it to False

Finish\_create = 1 // set it to True

Filter\_code = Generate random number;

Sigma = generate random Processing Time;

Phase = busy;

busy:

// Ignore the request;

}

λ (S) {

case phase

busy:

If Finish\_create = 1

//filter created

Send (Filter\_done, 1);

Send(Filter\_code, 1);

Passive: // Should not happen.

}

δint (S) {

case phase

If Finish\_create = 1

Finish\_create = 0 // set it to False

Start\_create = 1 // set it to True

busy:

phase = passive;

sigma = infinity;

passive: //never happens

}

Testing can be done by inputting one or more events at the input port and observing the outputs. The model should only respond to a 1 for the filter\_go input which is evident from the timestamps of the outputs shown below. A 1 should be received for filter\_done and a random number for the filter\_code.

00:00:10:00 in 1

00:00:30:00 in 1

00:00:50:00 in 0

00:00:60:00 in 1

00:00:70:00 in 0

00:00:80:00 in 0

00:00:90:00 in 0

00:00:100:00 in 1

00:00:110:00 in 0

00:00:120:00 in 0

00:00:130:00 in 1

00:00:140:00 in 1

00:00:148:00 in 0

00:00:150:00 in 0

Sample output is shown below.

00:00:13:975 filter\_done 1

00:00:31:592 filter\_done 1

00:01:01:915 filter\_done 1

00:01:44:070 filter\_done 1

00:02:12:365 filter\_done 1

00:02:24:321 filter\_done 1

The start\_module is also a sub-model consisting of 2 components: a StartCapture atomic model and a WirelessChannel atomic model, and is very similar to the previous coupled model. Together, these simulate the starting of a capture on a specific wireless channel (chosen randomly here). Both functions take a random amount of processing time to finish.

Start\_Module

Capture

Started

Capture

Started

Wireless\_Channel

Start\_Capture

Start Commandd

Start Command

Channel Done

Channel Go

Channel Number

Channel Number

Start\_module Formal specifications:

Start\_Module = <X,Y,D,EIC,EOC,IC,SELECT>

X = {Start Command}

Y = {Capture Started, Channel Number}

D={ Start\_Capture, Wireless\_Channel }

EIC = {( Start\_Module.Start Command, Start\_Capture.Start Command }

EOC = {( Start\_Capture.Capture Started, Start\_Module.Capture Started),

( Wireless\_Channel.Channel Number, Start\_Module.Channel\_Number)}

IC = {( Start\_Capture.Channel Go, Wireless\_Channel.Channel Go ) ,

(Wireless\_Channel.Channel Done, Start\_Capture.Channel Done)}

SELECT: Priority order (Descending) : Start\_Capture.Wireless\_Channel

Testing can be done by inputting one or more events at the input port – it should only respond to a 2 input evident from the timestamps of the outputs shown below – and verifying that a 1 is received for Capture Started and a random integer number for Channel Number.

00:00:10:00 in 2

00:00:30:00 in 3

00:00:50:00 in 1

00:00:60:00 in 3

00:00:90:00 in 2

Sample output is shown below.

00:00:15:567 channel\_num 0

00:00:17:482 capture\_started 1

00:01:36:435 channel\_num 5

00:01:40:756 capture\_started 1

Following is a description of the two atomic models which make up this coupled model.

The Start\_Capture is an atomic model which takes in two inputs as shown in the figure above and generates two outputs. It continuously waits for a new command to arrive at the input port and sends off the signal to Wireless\_Channel once a new Start Capture command is received. This sending off is simulated by generating a random processing time.

Upon receiving the Channel Done input, it proceeds to start the capture which is also simulated by generating a random processing time. It then sends the final Capture Started output to the output port.

Formal specifications for Start\_Capture = <X, Y, S, δint, δext, λ, ta>

X = {Start Command | Represented by integer value of 2, Channel Done}

Y = {Capture Started , Channel Go}

S = { Phase, sigma, wait\_for\_cmd, set\_channel, process\_cmd}

δext (S,e,X) {

Case port

Start\_Command:

Case Phase

Passive:

Check if wait\_for\_cmd = 1 and Command value is 2

Wait\_for\_cmd = 0 // set it to False

Set\_channel = 1 // set it to True

Sigma = generate random Processing Time;

Phase = busy;

busy:

// Ignore the request;

Channel\_Done:

Case Phase

Passive:

Check if set\_channel = 1

Set\_channel = 0 // set it to False

Process\_cmd = 1 // set it to True

Sigma = generate random Processing Time;

Phase = busy;

busy:

// Ignore the request;

}

λ (S) {

case phase

busy:

If Set\_channel = 1

//channel needs to be set, send the signal

Send (Channel\_go, 1);

If Process\_cmd = 1

// Capture started

Send (Capture\_started, 1);

Passive: // Should not happen.

}

δint (S) {

case phase

If Process\_cmd = 1

Process\_cmd = 0 // set it to False

Wait\_for\_cmd = 1 // Wait for new command

busy:

phase = passive;

sigma = infinity;

passive: //never happens

}

Start\_Capture was not tested individually, however, testing can be done by inputting one or more events – for each of the two input ports – and observing the outputs.

00:00:10:00 start\_command 2

00:00:50:00 channel\_done 1

00:00:60:00 start\_command 2

00:00:120:00 channel\_done 1

Wireless\_Channel simulates the setting of a channel by first generating a random number for the channel then waiting for a random processing time. It outputs a done signal and the channel number.

Formal specifications for Wireless\_Channel = <X, Y, S, δint, δext, λ, ta>

X = {Channel Go| Represented by a value of 1}

Y = {Channel Done , Channel Number}

S = { Phase, sigma, start\_set, finish\_set}

δext (S,e,X) {

Case port

Channel\_Go:

Case Phase

Passive:

Check if start\_set = 1 and Channel\_Go value is 1

Start\_set = 0 // set it to False

Finish\_set = 1 // set it to True

Channel\_number = Generate random number;

Sigma = generate random Processing Time;

Phase = busy;

busy:

// Ignore the request;

}

λ (S) {

case phase

busy:

If Finish\_set = 1

//channel set

Send (Channel\_done, 1);

Send(Channel\_number, 1);

Passive: // Should not happen.

}

δint (S) {

case phase

If Finish\_set = 1

Finish\_set = 0 // set it to False

Start\_set = 1 // set it to True

busy:

phase = passive;

sigma = infinity;

passive: //never happens

}

Testing can be done by inputting one or more events at the input port and observing the outputs. The model should only respond to a 1 for the channel\_go input – which is evident from the timestamps of the outputs shown below. A 1 should be received for channel\_done and a random integer number for the channel\_number.

00:00:10:00 in 1

00:00:30:00 in 1

00:00:50:00 in 0

00:00:60:00 in 1

00:00:70:00 in 0

00:00:80:00 in 0

00:00:90:00 in 0

00:00:100:00 in 1

00:00:110:00 in 0

00:00:120:00 in 0

00:00:130:00 in 1

00:00:140:00 in 1

00:00:148:00 in 0

00:00:150:00 in 0

Sample output is shown below.

00:00:13:975 channel\_done 1

00:00:31:592 channel\_done 1

00:01:01:915 channel\_done 1

00:01:44:070 channel\_done 1

00:02:12:365 channel\_done 1

00:02:24:321 channel\_done 1

The final component of the OmniEngine Control system is a simple atomic model named stop which simulates the stopping of a capture. It takes in one input and gives out a single output.

Capture Stopped

Stop Commandd

Stop\_Capture

Formal specifications for Stop\_Capture = <X, Y, S, δint, δext, λ, ta>

X = {Stop Command| Represented by a value of 3}

Y = {Capture Stopped}

S = { Phase, sigma, wait\_for\_cmd, process\_cmd}

δext (S,e,X) {

Case port

Stop\_Command:

Case Phase

Passive:

Check if wait\_for\_cmd = 1 and Stop\_Command value is 3

Wait\_for\_cmd = 0 // set it to False

Process\_cmd = 1 // set it to True

Sigma = generate random Processing Time;

Phase = busy;

busy:

// Ignore the request;

}

λ (S) {

case phase

busy:

If Process\_cmd = 1

//Capture stopped

Send (Capture\_Stopped, 1);

Passive: // Should not happen.

}

δint (S) {

case phase

If Process\_cmd = 1

Process\_cmd = 0 // set it to False

Wait\_for\_cmd = 1 // wait for new commands

busy:

phase = passive;

sigma = infinity;

passive: //never happens

}

Testing can be done by inputting one or more events at the input port and observing the outputs. The model should only respond to a 3 – which is evident from the timestamps of the outputs below - for the stop\_command input. A 1 should be received at the output.

00:00:10:00 in 2

00:00:30:00 in 3

00:00:50:00 in 1

00:00:60:00 in 3

00:00:90:00 in 2

Sample output is shown below.

00:00:33:975 capture\_stopped 1

00:01:01:592 capture\_stopped 1