A Guide to a Dedicated Event Graph Simulator for a Single Server System

October, 2013

Byoung K. Choi and Donghun Kang

Objective

This document provides a guide to the *single-server system event-graph simulator* presented in Section 4.7 of the textbook *Modeling and Simulation of Discrete-Event Systems*. It gives a technical description of how the dedicated event-graph simulator is implemented in C# language.

Recommendation

Prior to reading this document, the readers are recommended to read and understand Section 4.7 of the textbook. It is assumed that the reader has a basic working knowledge of C# (or Java). All source codes referred to in this document can be downloaded from the official website of the textbook (http://www.vms-technology.com/book).

History of This Document

Date	**Version**	**Reason**	**Person(s) in charge**
10/04/2013	1.0	Initial Draft	Donghun Kang <donghun.kang@kaist.ac.kr>
10/19/2013	1.1	Second Draft	Byoung K. Choi <bkchoi@kaist.ac.kr>

Table of Contents

1. Introduction

Consider a *single server system* consisting of an infinite-capacity buffer and a machine. In the single server system, a new job arrives every t_a minutes and is loaded on the machine if it is idle; otherwise the job is stored in the buffer. The loaded job is processed by the machine for t_s minutes and then unloaded. The freed machine loads another job from the buffer if it is not empty. The *inter-arrival time* t_a and the *service time* t_s are distributed as follows:

- Inter-arrival time: t_a ~ Expo(5)
- Service time: t_s ~ Uniform (4, 6)

We're going to collect the *average queue length* (AQL) statistics during the simulation.

1.1 The Event Graph Model

Figure 1 shows the event graph model of the single server system introduced in the textbook (See Fig.4.1-b in Section 4.2.2 of Chapter 4), where *Q* is the number of jobs in the buffer and M denotes the status of the machine (or the number of available machines in the system). Table 1 is the *event transition table* of the event graph model shown in Fig.1.

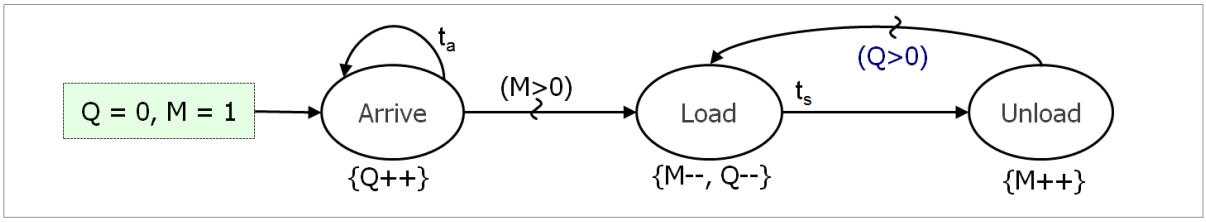

Fig.1. Event Graph Model of the Single Server System

Table 1. Event Transition Table of the Single Server System Event Graph Model in Fig. 1

No	Originating Event	State Change	Edge	Condition	Action	Delay	Destination Event
0	Initialize	Q = 0; M = 1;	1	True	schedule	0	Arrive
1	Arrive	Q++;	1	True	schedule	t_a= Exp(5)	Arrive
			2	M>0	schedule	0	Load
2	Load	M--; Q--;	1	True	schedule	t_s= Uni(4, 6)	Unload
3	Unload	M++;	1	Q>0	schedule	0	Load

1.2 Augmented Event Transition Table for Collecting Statistics

In order to collect the AQL statistics, the event transition table is augmented as follows:

① *Statistics variables* for collecting statistics are introduced:
- *SumQ*: sum of queue length Q over time
- *Before*: previous queue length change time of Q
- *AQL*: *average queue length* for the buffer

② SumQ and Before are initialized:
- SumQ = 0; Before = 0; // initialized at Initialize event

③ SumQ and Before are updated:
- SumQ += Q * (CLK – Before); Before = CLK; // updated at Arrive, Load events

④ *Statistics event* is newly introduced where AQL is computed:
- SumQ += Q * (CLK – Before); AQL = SumQ / CLK;

By incorporating the above additions, the augmented event transition table is obtained as in Table 2. More details on collecting statistics can be found at Section 4.7.5 of the textbook. In the following, Table 2 will be used in developing the dedicated simulator.

Table 2. Augmented Event Transition Table for collecting the average queue length

No	Originating Event	State Change	Edge	Condition	Delay	Destination Event
0	Initialize	Q= 0; M = 1; Before=0; SumQ=0	1	True	-	Arrive
1	Arrive	SumQ += Q*(CLK–Before); Before= CLK; Q++;	1	True	Exp(5)	Arrive
			2	M>0	0	Load
2	Load	SumQ += Q*(CLK–Before); Before= CLK; M--; Q--;	1	True	Uni(4,6)	Unload
3	Unload	M++;	1	Q>0	0	Load
4	Statistics	SumQ+= Q*(CLK–Before); AQL= SumQ/CLK				

2. Developing a Dedicated Event Graph Simulator

This section describes how a dedicated event graph simulator for the single server system is developed. C# codes are based on the pseudo codes given in Section 4.7 of the textbook.

2.1 Development Environment

The dedicated event graph simulator was developed with Microsoft Visual Studio 2010 and compiled with Microsoft .NET Framework Version 4.0. If you have Microsoft Visual Studio[1] 2010, please unzip the ***"singleserversystemsimulator.zip"*** file, which contains the source codes for the dedicated simulator and can be downloaded from the official site of the book (http://vms-technology.com/book/eventgraphsimulator), into a folder and open the solution file, which is named "***SingleServersystem.sln***".

2.2 Source Code Structure and Class Diagram

The project, named "*SingleServerSystem*", contains the *source code* which is composed of following files as depicted in Figure 2:

- `Simulator.cs`: *Simulator class* that contains a main program and event routines
- `Event.cs`: *Event class* that represents an event record
- `EventList.cs`: *EventList* class that implements the future event list (FEL)
- `Program.cs`: entry point of the program (do not modify this code)

[1] If you don't have Microsoft Visual Studio 2010, you can download a free version of Microsoft Visual Studio, named as Microsoft Visual C# 2010 Express or Microsoft Visual Studio Express 2012 for Windows Desktop. The Microsoft Visual Studio Express 2012 for Windows Desktop can be downloaded freely at the following URL:
http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-for-windows-desktop

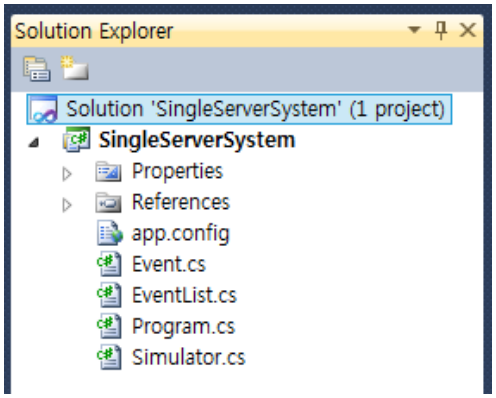

Fig.2. Source Code Structure shown in Solution Explorer of Visual Studio 2010

Figure 3 shows the class diagram consisting of three classes: *Simulator, EventList, and Event* classes. The *Simulator class* contains ***Main program*** (*Run*) together with Initialize & Statistics routines, ***event routines*** (*Arrive, Load, and Unload*), ***list-handling methods*** (*Schedule and Retrieve*) and ***random variate generators*** (*Exp and Uni*).

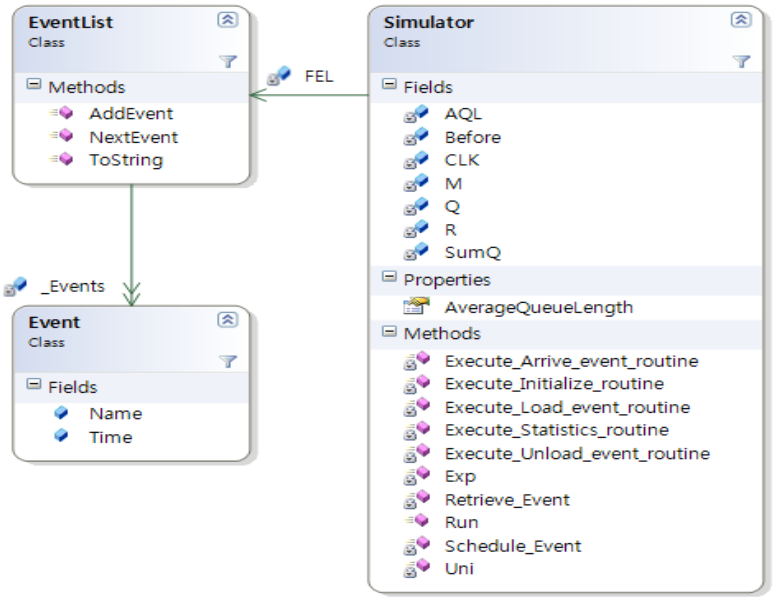

Fig.3. Class Diagram of the Dedicated Simulator

The ***member variables*** in the *Simulator* class include: (1) *state variables M, Q*; (2) simulation *clock CLK*; (3) *statistics variables SumQ, Before*, and *AQL*; (4) a *random number variable*, named *R*, for generating uniform random numbers which will be used in generating *Exp (m)* and *Uni (a, b)* random variates; and (5) the *event-list variable FEL*.

The *EventList class* contains methods for manipulating the *future event list FEL*, which is defined as a member variable of the *Simulator* class. The *Event class* is about the *next event* and has two *properties* of *Name* (event name) and *Time* (scheduled event time).

2.3 Main Program: Run method

The main program, whose pseudo-code was given in Fig. 4.59 (Section 4.7.5) of the textbook,

is implemented by the *Run method* as shown below. The main program consists of four phases: (1) *Initialization* phase, (2) *Time-flow mechanism* phase, (3) *Event-routine execution* phase, and (4) *Statistics collection* phase.

```
public void Run(double eosTime)
{
    //1. Initialization phase
    CLK = 0.0;

    FEL = new EventList();
    R = new Random();
    Event nextEvent = new Event();

    Execute_Initialize_routine(CLK);

    while (CLK < eosTime) {
        //2. Time-flow mechanism phase
        nextEvent = Retrieve_Event();
        CLK = nextEvent.Time;

        //3. Event-routine execution phase
        switch(nextEvent.Name) {
            case "Arrive": { Execute_Arrive_event_routine(CLK);break; }
            case "Load":   { Execute_Load_event_routine(CLK);break; }
            case "Unload": { Execute_Unload_event_routine(CLK);break; }
        }

        //Print out the event trajectory ("Time, Event Name, Q, M, FEL")
        Console.WriteLine("{0} \t{1} \t{2} \t{3} \t{4}", Math.Round(CLK,
                          2), nextEvent.Name, Q, M, FEL.ToString());
    }
        //4. Statistics collection phase
    Execute_Statistics_routine(CLK);
}
```

In the ***Initialization*** phase of the main program, (1) the *simulation clock* is set to zero, (2) member variables (*FEL* and *R*) and local variables (*nextEvent*) are declared, and (3) the initialization method *Execute_Initialize_routine ()* is invoked. As shown below, the initialization routine initializes the state variables (Q= 0; M= 1) and statistics variables (Before = 0; SumQ = 0) and schedules an initial event by invoking *Schedule_Event ("Arrive", Now)*.

```
private void Execute_Initialize_routine(double Now)
{
    //Initialize state variables
    Q = 0; M = 1;

    //Initialize statistics variables
    Before = 0; SumQ = 0;

    //Schedule Arrive event
    Schedule_Event("Arrive", Now);
}
```

In the ***Time-flow mechanism*** phase, a next event is retrieved by invoking the list-handling method *Retrieve_Event ()* and the simulation clock is updated; in the ***Event-routine execution*** phase, the *event routine* for each retrieved event is executed. Details of the event routines will

be given shortly. Finally, in the ***Statistics collection*** phase, the AQL (average queue length) is obtained by invoking the method Execute_Statistics_routine which is defined as below:

```
private void Execute_Statistics_routine(double Now)
{
    SumQ += Q * (Now - Before);
    AQL = SumQ / Now;
}
```

2.4 Event Routines

The *event-routine methods* in the Simulator class are:

(a) Execute_Arrive_event_routine (Now),

(b) Execute_Load_event_routine (Now), and

(c) Execute_Unload_event_routine (Now).

An event routine is a subprogram describing the changes in state variables and how the next events are scheduled and/or canceled for an originating event. One event routine is required for each event in an event graph and has the following structure: (1) Execute *state changes* and (2) schedule the *destination event* for each edge if the edge *Condition* is satisfied. The three event routine methods invoked by the main program are programmed in C# as follows. A next event is scheduled by invoking the list-handling method *Schedule_Event ()*.

```
private void Execute_Arrive_event_routine(double Now)
{
    SumQ += Q * (Now - Before); Before = Now;
    Q++;

    double ta = Exp(5);
    Schedule_Event("Arrive", Now + ta);
    if (M > 0)Schedule_Event("Load", Now);
}
```

```
private void Execute_Load_event_routine(double Now)
{
    SumQ += Q * (Now - Before); Before = Now;
    M--; Q--;

    double ts = Uni(4, 6);
    Schedule_Event("Unload", Now + ts);
}
```

```
private void Execute_Unload_event_routine(double Now)
{
    M++;
    if (Q >0)Schedule_Event("Load", Now);
}
```

2.5 List Handling Methods

As explained above, *EventList Class* implements the priority queue *FEL* (*future event list*) for managing next events. In the *Simulator* class, *FEL* was defined as a member variable as:

```
private EventList FEL;
```

There are two list-handling methods defined in the *Simulator class*: `Schedule_Event (name, time)` and `Retrieve_Event ()`. The `Schedule_Event` method is invoked at the *event routines* and the `Retrieve_Event` method is invoked at the *time-flow mechanism* phase of the main program. The two list-handling methods are defined as follows:

```
private void Schedule_Event(string name, double time)
{
    FEL.AddEvent(name, time);
}
```

```
private Event Retrieve_Event()
{
    Event nextEvent = null;
    nextEvent = FEL.NextEvent();
    return nextEvent;
}
```

There are two methods for manipulating the priority queue *FEL*: *AddEvent* and *NextEvent* methods. They are defined in the *EventList* class as follows:

- *AddEvent*(): adds an event to the list (sorted by the scheduled time of the event)
- NextEvent(): retrieves a next event next from the list

2.6 Random Variate Generators

Two random variates are defined at the *Simulator* class: Exponential and uniform random variates. A *uniform random variate* in the range of *a*, *b* is generated as follows:

```
private double Uni(double a, double b)
{
   if (a >= b) throw new Exception("The range is not valid.");
   double u = R.NextDouble();
   return (a + (b - a) * u);
}
```

R.NextDouble () method returns a random number between 0.0 and 1.0. As mentioned in Section 2.2, "R" is a member variable of the *Simulator* class, which is a pseudo-random number generator (*System.Random* class) provided by C# language.

```
private Random R;
```

The exponential random variate is generated using the *inverse transformation method* given in Section 3.4.2 of the textbook. Math.Log () method returns the natural logarithm.

```
private double Exp(double a)
{
    if (a <= 0)
        throw new ArgumentException("Negative value is not allowed");
    double u = R.NextDouble();
    return (-a * Math.Log(u));
}
```

3. Simulation Execution

If you want to run the dedicated simulator from Visual Studio 2010, click the menu item *Debug* > *Start Without Debugging* (or click the short key, Ctrl + F5) as shown in Figure 4. Then, *system trajectory* values and *average queue length* are printed on the console as shown in Figure 5. Also, you can run the dedicated simulator from the file system: an executable file "SingleServerSystem.exe" is provided under a folder of "SingleServerSystem\bin\Debug".

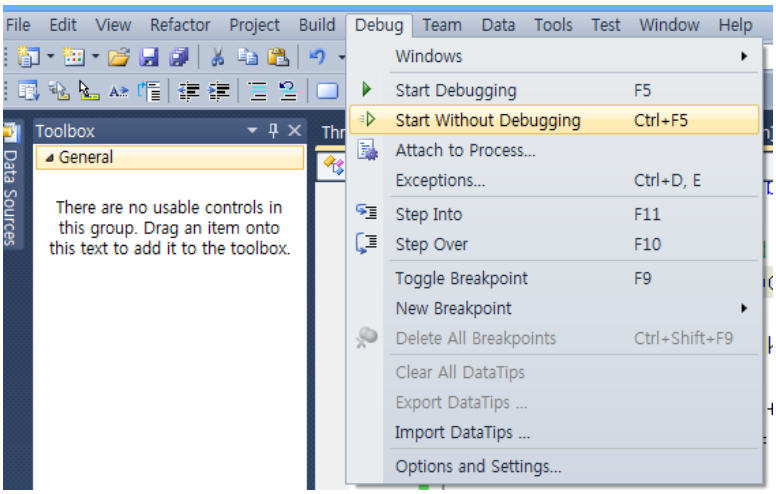

Fig. 4. Run the Dedicated Simulator from Visual Studio

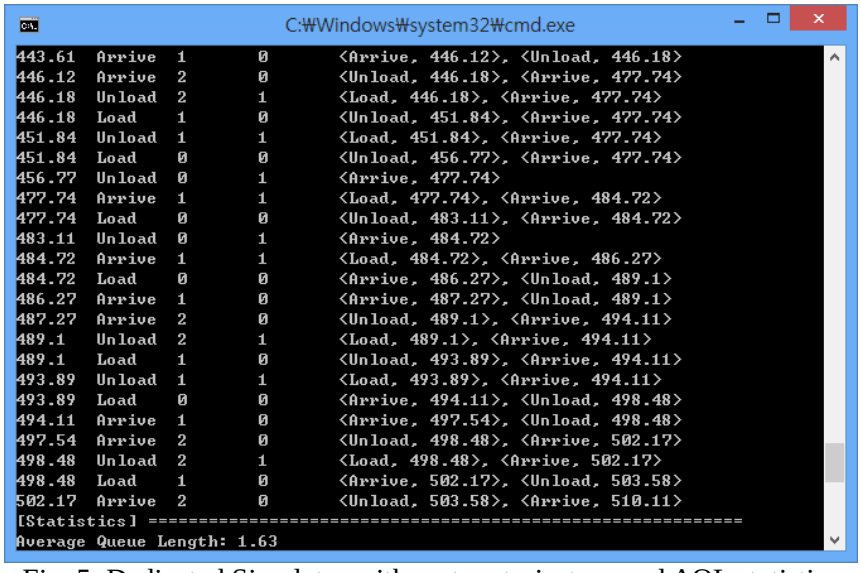

Fig. 5. Dedicated Simulator with system trajectory and AQL statistics

This is a kind of prototype implementation so that the readers may develop their own dedicated simulator (equipped with GUI) from it. As shown in Fig. 5, the system trajectory consists of five columns: Time, Event Name, Q, M, and FEL. The last column FEL shows the scheduled events (name and time) stored in the FEL.

4. Source Codes

In this section, the source codes of the single server system event graph simulator are provided: `Event.cs` for *Event* class, `EventList.cs` for EventList class, and `Simulator.cs` for Simulator class.

4.1 Event.cs

```
using System;
using System.Text;

namespace MSDES.Chap04.SingleServerSystem {
    /// <summary>
    /// Class for an Event Record
    /// </summary>
    public class Event {
        #region Member Variables
        private string _Name;
        private double _Time;
        #endregion

        #region Properties
        /// <summary>
        /// Event Name
        /// </summary>
        public string Name { get { return _Name; } }

        /// <summary>
        /// Event Time
        /// </summary>
        public double Time { get { return _Time; } }
        #endregion

        #region Constructors
        public Event() { }

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="name">the name of an event</param>
        /// <param name="time">the time of an event</param>
        public Event(string name, double time) {
            _Name = name;
            _Time = time;
        }
        #endregion

        #region Methods
        public override bool Equals(object obj)  {
            bool rslt = false;
            Event target = (Event)obj;
            if (target != null && target.Name == _Name &&
                target.Time == _Time)
                rslt = true;

            return rslt;
        }

        public override string ToString() {
            return _Name + "@" + _Time;
        }
```

```
        public override int GetHashCode() {
            return ToString().GetHashCode();
        }
        #endregion
    }
}
```

4.2 EventList.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MSDES.Chap04.SingleServerSystem {
    public class EventList {
        #region Member Variables
        private List<Event> _Events; // future event list
        #endregion

        #region Constructors
        public EventList(){
            _Events = new List<Event>();
        }
        #endregion

        #region Methods
        public void Initialize() {
            _Events.Clear();
        }

        /// <summary>
        /// Schedule an event into the future event list (FEL)
        /// </summary>
        /// <param name="eventName">Event Name</param>
        /// <param name="eventTime">Event Time</param>
        public void AddEvent(String eventName, double eventTime)  {
            Event nextEvent = new Event(eventName, eventTime);
            if (_Events.Count == 0) {
                _Events.Add(nextEvent);
            } else {
                bool isAdded = false;
                for (int i = 0; i < _Events.Count; i++) {
                    Event e = _Events[i];
                    if (nextEvent.Time <= e.Time) {
                        _Events.Insert(i, nextEvent);
                        isAdded = true;
                        break;
                    }
                }
                if (!isAdded)
                    _Events.Add(nextEvent);
            }
        }

        /// Return an event record that located at the first element
        /// in the future event list(FEL).
        /// </summary>
        /// <returns>An event record</returns>
        public Event NextEvent() {
            Event temp_event = null;
```

```
            if (_Events.Count > 0) {
                temp_event = _Events[0];
                _Events.RemoveAt(0);
            }
            return temp_event;
        }

        /// <summary>
        /// Cancel an event which has the same name of a given name
        /// from the FEL.
        /// </summary>
        /// <param name="eventName">Event Name</param>
        public void RemoveEvent(String eventName) {
            Event CancelEvent = null;
            for (int i = 0; i < _Events.Count; i++) {
                Event e = _Events[i];
                if (e.Name == eventName) {
                    CancelEvent = e; break;
                }
            }
            if (CancelEvent != null)
                _Events.Remove(CancelEvent);
        }

        /// <summary>
        /// Make a string that contains the information of all event
        /// records of the FEL
        /// </summary>
        public override string ToString() {
            string fel = "";

            for (int i = 0; i < _Events.Count; i++) {
                if (i != 0)
                    fel += ", ";
                fel += "<" + _Events[i].Name + ", " +
                         Math.Round(_Events[i].Time, 2) + ">";
            }
            return fel;
        }
        #endregion
    }
}
```

4.3 Simulator.cs

```
using System;
using System.Text;

namespace MSDES.Chap04.SingleServerSystem {
    public class Simulator {
        #region Member variables for state variables
        /// <summary>
        /// Number of available machines
        /// </summary>
        private int M;
        /// <summary>
        /// Number of jobs awaiting at the buffer
        /// </summary>
        private double Q;
        #endregion
```

```
        #region Member Variables for Simulator objects
        /// <summary>
        /// Simulation Clock
        /// </summary>
        private double CLK;
        /// <summary>
        /// Future Event List
        /// </summary>
        private EventList FEL;
        #endregion

        #region Member Variables for Statistics
        private double Before;
        private double SumQ;
        private double AQL;
        #endregion

        /// <summary>
        /// Pseudo Random Value Generator
        /// </summary>
        private Random R;

        #region Properties
        /// <summary>
        /// Average Queue Length at the Buffer
        /// </summary>
        public double AverageQueueLength { get { return AQL; } }
        #endregion

        #region Constructors
        public Simulator(){ }
        #endregion

        #region Methods for Main Program
        /// <summary>
        /// Run the simulation using next-event scheduling algorithm
        /// </summary>
        public void Run(double eosTime) {
            //1. Initialization phase
            CLK = 0.0;
            //Initialize the FEL
            FEL = new EventList();
            //Initialize Random variate R
            R = new Random();

            Execute_Initialize_routine(CLK);

            Event nextEvent = new Event();
            while (CLK < eosTime) {
                //2. Time-flow mechanism phase
                nextEvent = Retrieve_Event();
                CLK = nextEvent.Time;

                //3. Event-routine execution phase
                switch(nextEvent.Name) {
                    case "Arrive": {
                        Execute_Arrive_event_routine(CLK);break; }
                    case "Load":   {
                        Execute_Load_event_routine(CLK);break; }
                    case "Unload": {
                        Execute_Unload_event_routine(CLK);break; }
                }
```

```
            //Print out the event trajectory "Time, Name, Q, M, FEL"
            Console.WriteLine("{0} \t{1} \t{2} \t{3} \t{4}",
              Math.Round(CLK, 2), nextEvent.Name, Q, M, FEL.ToString());
        }

        //4. Statistics calculation phase
        Execute_Statistics_routine(CLK);
    }
    #endregion

    #region Methods for Handling Events
    /// <summary>
    /// Schedule an event into the future event list (FEL)
    /// </summary>
    /// <param name="name">Event Name</param>
    /// <param name="timeime">Event Time</param>
    private void Schedule_Event(string name, double time) {
        FEL.AddEvent(name, time);
    }

    /// <summary>
    /// Return an event record that located at the first element
    /// in the future event list(FEL).
    /// </summary>
    /// <returns>An event record</returns>
    private Event Retrieve_Event() {
        Event nextEvent = null;
        nextEvent = FEL.NextEvent();
        return nextEvent;
    }

    /// <summary>
    /// Cancel an event which has the same name of a given name
    /// from the FEL.
    /// </summary>
    /// <param name="eventName">Event Name</param>
    private void Cancel_Event(string eventName) {
        FEL.RemoveEvent(eventName);
    }

    #endregion

    #region Event Routines
    /// <summary>
    /// Execute initialize routine
    /// </summary>
    /// <param name="Now"> Time </param>
    private void Execute_Initialize_routine(double Now) {
        //Initialize Q, and M (state variables)
        Q = 0;
        M = 1;

        //Initialize the state variables for collecting statistics
        Before = 0; SumQ = 0;

        //Schedule Arrive event
        Schedule_Event("Arrive", Now);
    }

    /// <summary>
    /// Execute Arrive event routine
    /// </summary>
    /// <param name="Now">Current Simulation Clock</param>
```

```
        private void Execute_Arrive_event_routine(double Now) {
            SumQ += Q * (Now - Before); Before = Now;
            Q++;

            double ta = Exp(5);
            Schedule_Event("Arrive", Now + ta);

            if (M > 0)
                Schedule_Event("Load", Now);
        }

        /// <summary>
        /// Execute Load event routine
        /// </summary>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Load_event_routine(double Now) {
            SumQ += Q * (Now - Before); Before = Now;
            M--;
            Q--;

            double ts = Uni(4, 6);
            Schedule_Event("Unload", Now + ts);
        }

        /// <summary>
        /// Execute Unload event routine
        /// </summary>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Unload_event_routine(double Now) {
            M++;

            if (Q >0)
                Schedule_Event("Load", Now);
        }

        /// <summary>
        /// Execute Statistics routine after the simulation clock reaches
the end of simulation time
        /// </summary>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Statistics_routine(double Now) {
            SumQ += Q * (Now - Before);
            AQL = SumQ / Now;
        }

        #endregion

        #region Methods for Generating Random Variates
        /// <summary>
        /// Generate a random value which follows the exponential
        /// distribution with a given mean of a
        /// </summary>
        /// <param name="a">A mean value</param>
        /// <returns>Exponential Random value </returns>
        private double Exp(double a) {
            if (a<=0) throw
                new ArgumentException("Negative value is not allowed");
            double u = R.NextDouble();
            return (-a * Math.Log(u));
        }

        /// <summary>
        /// Generate a random value which follows the uniform
```

```
        /// distribution with a given range of a and b
        /// </summary>
        /// <param name="a">Start range</param>
        /// <param name="b">End range</param>
        /// <returns></returns>
        private double Uni(double a, double b) {
            if (a>=b) throw new Exception("The range is not valid.");
            double u = R.NextDouble();
            return (a + (b - a) * u);
        }
        #endregion
    }
}
```