A Guide to a Dedicated PEG Simulator for a Three-stage Tandem Line

October, 2013

Byoung K. Choi and Donghun Kang

Objective

This document provides a guide to the *three-stage tandem line parameterized event-graph (PEG) simulator* presented in Section 5.7 of the textbook, *Modeling and Simulation of Discrete-Event Systems*. It gives a technical description of how the dedicated PEG simulator is implemented in C# language.

Recommendation

Prior to reading this document, the readers are recommended to read and understand Section 5.7 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>
10/29/2013	1.1	Second Draft (with C# codes)	Byoung K. Choi <bkchoi@kaist.ac.kr>

Table of Contents

1. Introduction

Consider a *tandem line* consisting of three *stages* connected in tandem where each stage consists of an infinite-capacity *buffer* and a *machine*. This tandem line is called a *3-stage tandem line*. Jobs are generated at an *inter-interval time* of t_a minutes, and each job goes through *stages k* for k=1, 2, 3. The job *service time at stage k* is *t[k]*. It is assumed the distributions of the inter-arrival time and service times are as follow:

- Inter-arrival time: *ta* ~ Expo(10)
- Service time at stage 1: *t[1]* ~ Uniform (10, 15)
- Service time at stage 2: *t[2]* ~ Uniform (13, 18)
- Service time at stage 3: *t[3]* ~ Uniform (8, 13)

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

1.1 Parameterized Event Graph (PEG) Model and Event Transition Table

Figure 1 shows the *PEG model* of the 3-stage tandem line introduced in the textbook (See Fig.5.36 in Section 5.6.3 of Chapter 5), where Q[k] and M[k] denote the numbers of jobs in the buffer and of available machines at stage k, respectively. Table 1 shows the *event transition table* of the PEG model in Fig.1.

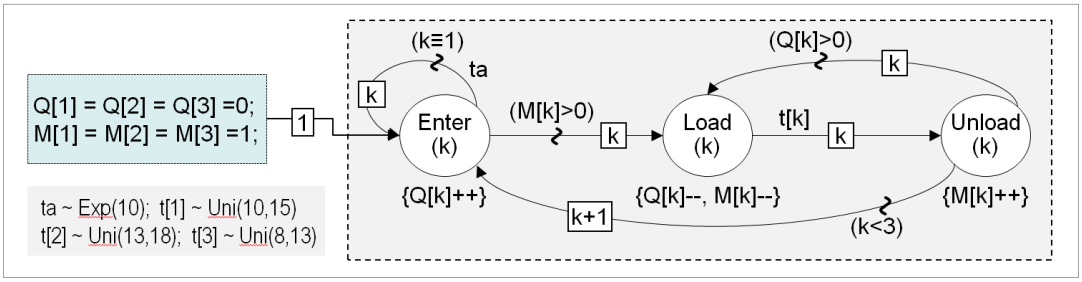

Fig.1 PEG Model of the Three-stage Tandem Line

Table 1 Event Transition Table of the 3-stage Tandem Line PEG Model in Fig.1

No	Event	State Change	Edge	Condition	Delay	Parameter	Destination Event
0	Initialize	For k=1~3 {Q[k] = 0; M[k] = 1}	1	True	0	1	Enter (k)
1	Enter(k)	Q[k]++; if (k==1) ta= Exp(10);	1	k ≡ 1	ta	k	Enter (k)
			2	M[k] > 0	0	k	Load (k)
2	Load(k)	Q[k]--; M[k]--; t[k]= (k≡1)*Uni(10,15)+ (k≡2)*Uni(13,18)+ (k≡3)*Uni(8,13);	1	True	t[k]	k	Unload (k)
3	Unload(k)	M[k]++;	1	Q[k] > 1	0	k	Load (k)
			2	k< 3	0	k+1	Enter (k)

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 Additional state variables for collecting statistics are introduced:

- SumQ[k]: sum of queue lengths Q[k] over time at stage k
- Before[k]: previous change time of Q[k]

- AQL[k]: average queue length for each stage k

② SumQ[k] and Before[k] are initialized:

- SumQ[k] = 0; Before[k] = 0; // initialized at Initialize event

③ SumQ[k] and Before[k] are updated:

- SumQ[k]+= Q[k]*(CLK–Before[k]); Before[k]= CLK; // updated at Enter[k], Load[k] events

④ Statistics event is newly introduced where AQL[k] is computed:

- For k=1~3 {SumQ[k] += Q[k] * (CLK – Before[k]); AQL[k] = SumQ[k] / CLK;}

By incorporating the above additions, the augmented event transition table is obtained as in Table 2. In the following, Table 2 will be used in developing a dedicated PEG simulator.

Table 2. Augmented Event Transition Table for computing average queue lengths

No	Event	State Change	Edge	Condition	Delay	Parameter	Next Event
0	Initialize	For k=1~3 {Q[k]=0; M[k]=1; Before[k]=0; SumQ[k]=0}	1	True	-	1	Enter(k)
1	Enter(k)	SumQ[k] += Q[k]*(CLK–Before[k]); Before[k] = CLK; Q[k]++; if (k==1) ta= Exp(10);	1	k ≡ 1	ta	k	Enter(k)
			2	M[k]>0	0	k	Load(k)
2	Load(k)	SumQ[k]+= Q[k]*(CLK–Before[k]); Before[k]= CLK; Q[k]--; M[k]--; t[k]= (k≡1)*Uni(10,15)+ (k≡2)*Uni(13,18)+ (k≡3)*Uni(8,13);	1	True	t[k]	k	Unload(k)
3	Unload(k)	M[k]++;	1	Q[k]>0	0	k	Load(k)
			2	k<3	0	k+1	Enter(k)
4	Statistics	For k=1~3 { SumQ[k] += Q[k]*(CLK–Before[k]); AQL[k]= SumQ[k]/CLK; }					

2. Developing a Dedicated PEG Simulator

This section describes how a dedicated PEG simulator for a three-stage tandem line is developed. C# codes are based on the pseudo codes given in Section 5.7 of the textbook.

2.1 Development Environment

The dedicated PEG 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 ***"threestagetandemlinepegsimulator.zip"*** file, which contains the source codes for the dedicated PEG 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 "***Three-stage Tandem Line Simulator.sln***".

2.2 Source Code Structure and Class Diagram

The project, named "*Three-stage Tandem Line Simulator*", contains the *source code* which is composed of following files as depicted in Figure 2:

- `Simulator.cs`: consists of *Simulator* class that has a main program and event

[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 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

routines

- `Event.cs`: consists of *Event* class that represents an event record
- `EventList.cs`: consists of *EventList* class that implements the future event list
- `MainFrm.cs`: consists of the graphical user interface for the simulator
- `Program.cs`: entry point of the program (do not modify this code)

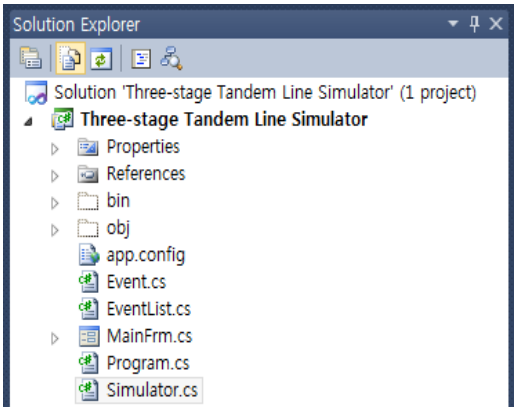

Fig. 2. Source Code Structure shown in Solution Explorer of Visual Studio 2010

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

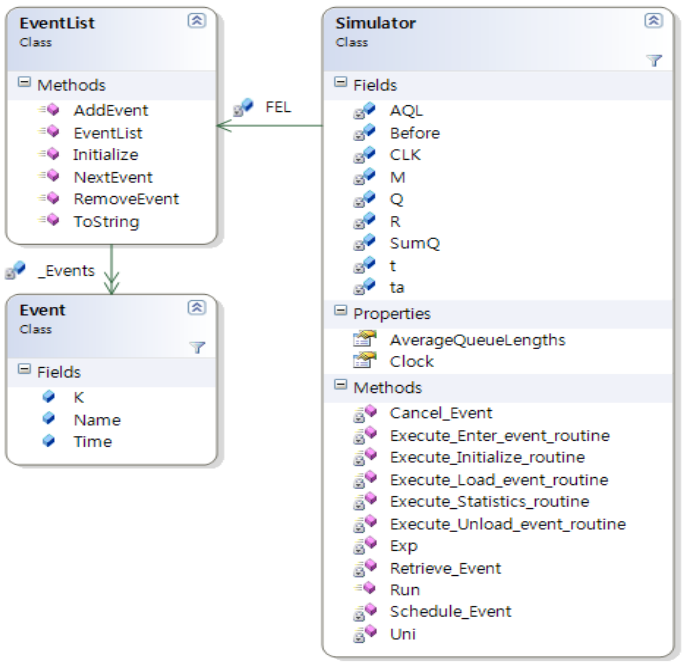

Fig. 3. Class Diagram of the Dedicated Simulator

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) *time variables t[]* and *ta*; (5) 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 (6) 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* deals with the *next event* and has three *properties* of *K* (index), *Name* (event name) and *Time* (event time).

2.3 Main Program (Run method)

The main program, whose structure was given in Fig. 5.49 (Section 5.7.1) of the textbook, is implemented by the *Run method* as shown below. The main program consists of four phases: (1) Initialization, (2) Time-flow mechanism, (3) Event-routine execution, and (4) Statistics calculation.

```
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 "Enter": { Execute_Enter_event_routine(nextEvent.K, CLK);
                         break; }
         case "Load": { Execute_Load_event_routine(nextEvent.K, CLK);
                        break; }
         case "Unload": { Execute_Unload_event_routine(nextEvent.K, CLK);
                        break; }
      }
   }

   //4. Statistics calculation 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 method *Execute_Initialize_routine()* is invoked. As shown below, the initialization method initializes state variables (Q[] = 0; M[] = 1) and statistics variables (Before[] = 0; SumQ[] = 0) and schedules an initial event by invoking *Schedule_Event ("Enter", Now)*.

```
private void Execute_Initialize_routine(double Now)
{
    //Initialize the state variables & statistics variables
    Q = new int[4]; M = new int[4];
    Before = new double[4]; SumQ = new double[4];

    for (int k = 1; k <= 3; k++) {
        Q[k] = 0; M[k] = 1;
        Before[k] = 0; SumQ[k] = 0;
    }

    //Schedule Enter event
    Schedule_Event("Enter", 1, 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, AQL[k] (average queue lengths for k= 1, 2, 3) is obtained by invoking the method Execute_Statistics_routine which is defined as below:

```
private void Execute_Statistics_routine(double Now)
{
    AQL = new double[4];
    for (int k = 1; k <= 3; k++) {
        SumQ[k] += Q[k] * (Now - Before[k]);
        AQL[k] = SumQ[k] / Now;
    }
}
```

2.4 Event Routines

The *event-routine methods* in the Simulator class are ('k' is the parameter):

(a) `Execute_Arrive_event_routine (k, Now)`,

(b) `Execute_Load_event_routine (k, Now)`, and

(c) `Execute_Unload_event_routine (k, 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_Enter_event_routine(int k, double Now)
{
    SumQ[k] += Q[k] * (Now - Before[k]); Before[k] = Now;
    Q[k]++;

    if (k == 1) {
        double ta = Now + Exp(3);
        Schedule_Event("Enter", k, ta);
    }
     if (M[k] > 0)
```

```
            Schedule_Event("Load", k, Now);
    }
```

```
private void Execute_Load_event_routine(int k, double Now)
{
    SumQ[k] += Q[k] * (Now - Before[k]); Before[k] = Now;
    Q[k]--; M[k]--;

    double ts = (k == 1 ? 1 : 0) * Uni(10, 15) +
                (k == 2 ? 1 : 0) * Uni(13, 18) +
                (k == 3 ? 1 : 0) * Uni(8, 13);
    Schedule_Event("Unload", k, Now + ts);
}
```

```
private void Execute_Unload_event_routine(int k, double Now)
{
    M[k]++;
    if (Q[k] > 0)
        Schedule_Event("Load", k, Now);
    if (k < 3)
        Schedule_Event("Enter", k + 1, 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, k, 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, int k, double time)
{
    FEL.AddEvent(name, k, 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). You can also run the dedicated simulator from the file system: you can find an executable file, "TandemLineSimulator.exe" under a folder of "TandemLinePEGSimulator\bin\Debug".

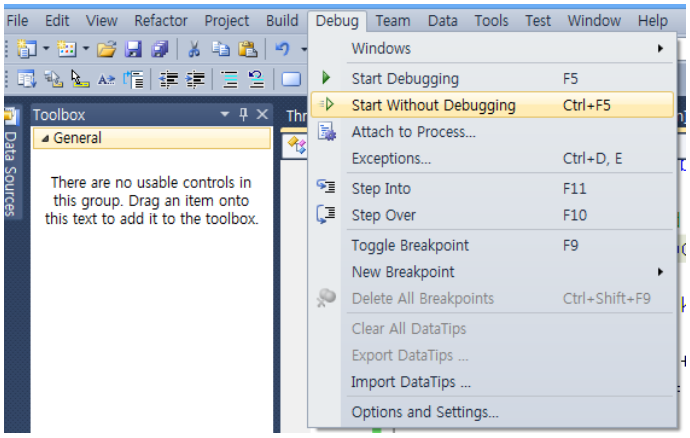

Fig. 4. Run the Dedicated Simulator from Visual Studio

If you run the dedicated simulator by clicking "Run" button, you can see the following

window that shows the system trajectory (on the top part), average queue length (on the bottom-left part), and chart for queue length changes over time (on the bottom-right part):

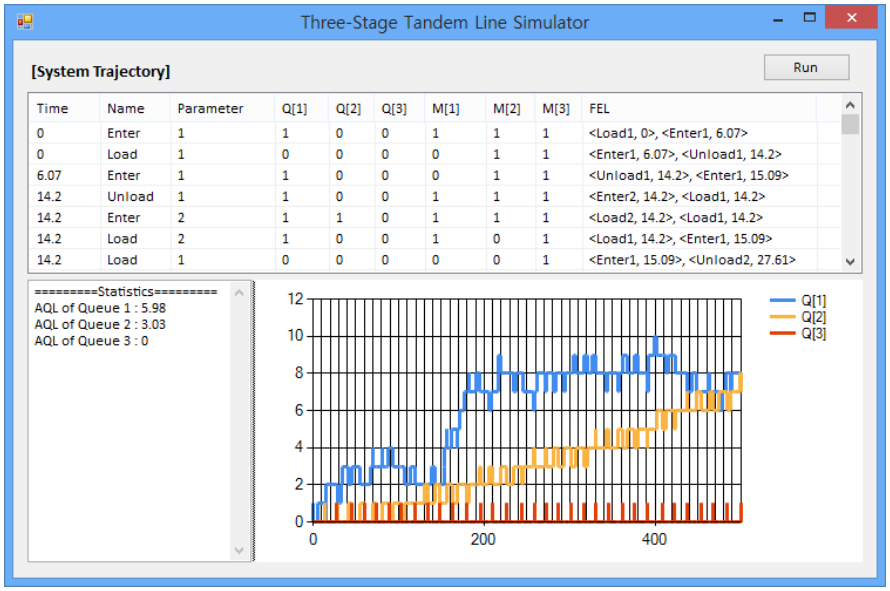

Fig. 5. Dedicated Simulator with system trajectory and AQL statistics

In the System Trajectory, you can observe how the system states change over time. First three columns are Time, Name, and Parameter where an event (Name) is occurred at a Time with Parameter. The following six columns represent the values of *Q[k]* and *M[k]* for k that varies from 1 to 3. And, the last column shows the contents of the future event list at the specified Time.

Displayed in the AQL (average queue length) part located at the bottom-left of Fig.5 are AQLs for the three queues Q[k] for k=1, 2, 3. Displayed right to the AQL part are queue-length graphs for the three queues. Observe that (1) the queue lengths at stages 1 and 2 increase steadily (because arrival rates are higher than service rates) and (2) the queue lengths at stage 3 are at most one (because the service time at stage 3 is smaller than that at stage 2).

4. Source Codes

In this section, the source codes of the three-stage tandem line 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.Chap05.TandemLine
{
    /// <summary>
    /// Class for an Event Record
    /// </summary>
    public class Event {
```

```
        #region Member Variables
        private string _Name;
        private int _K;
        private double _Time;
        #endregion

        #region Properties
        public string Name { get { return _Name; } }
        public int K { get { return _K; } }
        public double Time { get { return _Time; } }
        #endregion

        /// <summary>
        /// Parameterless default constructor
        /// </summary>
        public Event() { }

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

        /// <summary>
        /// Check if the parameter is equal to the event
        /// </summary>
        /// <param name="obj">Object to compare with this event</param>
        public override bool Equals(object obj) {
            if (obj is Event) {
                Event target = obj as Event;
                if (target.Name == this.Name && target.K == this.K
                     && target.Time == this.Time)
                    return true;
                else
                    return false;
            } else
                return false;
        }
    }
}
```

4.2 EventList.cs

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

namespace MSDES.Chap05.TandemLine {
    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, int eventParameter,
                             double eventTime) {
            Event nextEvent =
                      new Event(eventName, eventParameter, 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>
        /// Remove an event record that has the same name of a given name
with the same parameter.
        /// </summary>
        /// <param name="eventName">Event Name to be Canceled</param>
        public void RemoveEvent(string eventName, int eventParameter) {
            Event CancelEvent = null;
            for (int i = 0; i < _Events.Count; i++) {
                Event e = _Events[i];
                if (e.Name == eventName && e.K == eventParameter) {
                    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 + _Events[i].K.ToString() + ",
" + Math.Round(_Events[i].Time, 2) + ">";
            }
            return fel;
        }
        #endregion
    }
}
```

4.3 Simulator.cs

```
using System;
using System.Text;

namespace MSDES.Chap05.TandemLine {
    public class Simulator {
        #region Member Variables for State Variables
        /// <summary>
        /// Machine Status of each stage
        /// </summary>
        private int[] M;
        /// <summary>
        /// Number of jobs waiting at each stage
        /// </summary>
        private int[] Q;
        #endregion

        #region Member Variables of Simulation 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

        #region Member Variables for Arrival Time and Service Times
        /// <summary>
```

```
        /// Arrival Time
        /// </summary>
        private double ta;
        #endregion

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

        #region Properties
        /// <summary>
        /// Current Simulation Clock
        /// </summary>
        public double Clock { get { return this.CLK; } }

        /// <summary>
        /// Average queue length at each stage
        /// </summary>
        /// <returns>the average queue length</returns>
        public double[] AverageQueueLengths { 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();
            //Initialize next event
            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 "Enter": {
                     Execute_Enter_event_routine(nextEvent.K, CLK); break;}
                    case "Load": {
                     Execute_Load_event_routine(nextEvent.K, CLK); break;}
                    case "Unload": {
                     Execute_Unload_event_routine(nextEvent.K, CLK); break;}
                }
                //Print out the event trajectory "Time, Name, Parmeter, Q,
                // M, FEL" and the graph of Queues
                MainFrm.App.AddTrajectory(Math.Round(nextEvent.Time,2),
                  nextEvent.Name, nextEvent.K, Q, M, FEL.ToString());
                MainFrm.App.AddChart(CLK, Q);
```

```
            }
            //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="k">Event Parameter</param>
        /// <param name="time">Event Time</param>
        private void Schedule_Event(string name, int k, double time) {
            FEL.AddEvent(name, k, time);
        }

        /// <summary>
        /// Return an event record that located at the first element
        /// in the 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 and parameter of
        /// given name and k from the FEL.
        /// </summary>
        /// <param name="name">Event Name</param>
        /// <param name="k">Event Parameter</param>
        private void Cancel_Event(String name, int k) {
            FEL.RemoveEvent(name, k);
        }
        #endregion

        #region Event Routines
        /// <summary>
        /// Execute initialize routine
        /// </summary>
        /// <param name="Now"> Time </param>
        private void Execute_Initialize_routine(double Now) {
            //Initialize the state variables
            Q = new int[4]; M = new int[4];
            Before = new double[4]; SumQ = new double[4];
            for (int k = 1; k <= 3; k++) {
                Q[k] = 0; M[k] = 1;
                Before[k] = 0; SumQ[k] = 0;
            }

            //Schedule Enter event
            Schedule_Event("Enter", 1, Now);
        }

        /// <summary>
        /// Execute Enter event routine
        /// </summary>
        /// <param name="j">Parameter of an event</param>
        /// <param name="Now">Current Simulation Clock</param>
```

```
        private void Execute_Enter_event_routine(int k, double Now) {
            SumQ[k] += Q[k] * (Now - Before[k]); Before[k] = Now;
            Q[k]++;

            if (k == 1) {
                ta = Now + Exp(10);
                Schedule_Event("Enter", k, ta);
            }

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

        /// <summary>
        /// Execute Load event routine
        /// </summary>
        /// <param name="j">Parameter of an event</param>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Load_event_routine(int k, double Now) {
            SumQ[k] += Q[k] * (Now - Before[k]); Before[k] = Now;
            Q[k]--; M[k]--;

            double ts = (k == 1 ? 1 : 0) * Uni(10, 15) + (k == 2 ? 1 : 0)
                        * Uni(13, 18) + (k == 3 ? 1 : 0) * Uni(8, 13);
            Schedule_Event("Unload", k, Now + ts);
        }

        /// <summary>
        /// Execute Unload event routine
        /// </summary>
        /// <param name="k">Parameter of an event</param>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Unload_event_routine(int k, double Now) {
            M[k]++;
            if (Q[k] > 0)
                Schedule_Event("Load", k, Now);
            if (k < 3)
                Schedule_Event("Enter", k + 1, Now);
        }

        /// <summary>
        /// Calculate the average queue length
        /// </summary>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Statistics_routine(double Now) {
            AQL = new double[4];
            for (int k = 1; k <= 3; k++)
            {
                SumQ[k] += Q[k] * (Now - Before[k]);
                AQL[k] = SumQ[k] / Now;
            }
        }
        #endregion

        #region Random-variate Generation Methods
        /// <summary>
        /// Returns a random value that 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 Exception("Negative value is not allowed");
            double u = R.NextDouble();
            return (-a * Math.Log(u));
        }

        /// <summary>
        /// Returns a random value that 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>Uniform random value</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
    }
}
```