A Guide to a Dedicated *Parameterized Event Graph Simulator* for a Simple Job Shop

November, 2013

Byoung K. Choi and Donghun Kang

Objective

This document provides a guide to the *simple job shop PEG (shop parameterized event graph) simulator* presented in Section 5.7.2 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 5.7.2 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**	**Persons in charge**
11/19/2013	1.0	Initial Draft	Donghun Kang <donghun.kang@kaist.ac.kr>
11/29/2013	1.1	Final Draft	Byoung K. Choi <bkchoi@kaist.ac.kr>

Table of Contents

1. Introduction 1

1.1 The Parameterized Event Graph (PEG) Model and Event Transition Table 2

1.2 Augmented Event Transition Table for Collecting Statistics 3

2. Developing a Dedicated PEG Simulator 4

2.1 Development Environment 4

2.2 Source Code Structure and Class Diagram 4

2.3 Main Program (Run method) 5

2.4 Event Routines 7

2.5 List Handling Methods 9

2.6 Random Variate Generators 10

3. Simulation Execution 10

4. Source Codes 11

4.1 Event.cs 11

4.2 EventList.cs 13

4.3 Simulator.cs 15

1. Introduction

Consider a *simple job shop* consisting of four ***stations*** (s = 0, 1, 2, 3) with each station having one machine (i.e., M[s] = 1). There are three ***job types*** (j = 0, 1, 2) with each job type having its own unique sequence of ***processing steps*** (p = 1, 2 …). The station number (s) of a job type (j) for a processing step (p) is specified in the ***routing sequence*** of the job type. A job may visit a station more than once. A job arrives at every 12 minutes (t_a = 12) with ***job-mix*** *ratios* of 26% for j = 0, 48% for j = 1, and 26% for j = 2. The routing sequences route(j,p) and ***processing times*** {t_{jp}} are given in Table 1. All machines need a setup time of 30 minutes (t_s = 30) whenever there is a job-type change.

Table 1. Routing sequence {route(j,p)} and processing times

	Step-0 (p=0)		Step-1 (p=1)		Step-2 (p=2)		Step-3 (p=3)		Step-4 (p=4)	
Job (Ratio)	route(j,0)	t_{j0}	route(j,1)	t_{j1}	route(j,2)	t_{j2}	route(j,3)	t_{j3}	route(j,4)	t_{j4}
j=0 (26%)	0	Exp(6)	1	Exp(5)	2	Exp(15)	3	Exp(8)	-	-
j=1 (48%)	0	Exp(11)	1	Exp(4)	3	Exp(15)	1	Exp(6)	2	Exp(27)
j=2 (26%)	1	Exp(7)	0	Exp(7)	2	Exp(18)		-		-

The routing sequence of type-1 jobs is depicted in Figure 1 where processing times (t_{1p} for p = 0, 1, 2, 3) and transport delay times (d_{vw}) from station 'v' to station 'w' are also shown. The ***transport delay times*** delay[v, w] are summarized in Table 2.

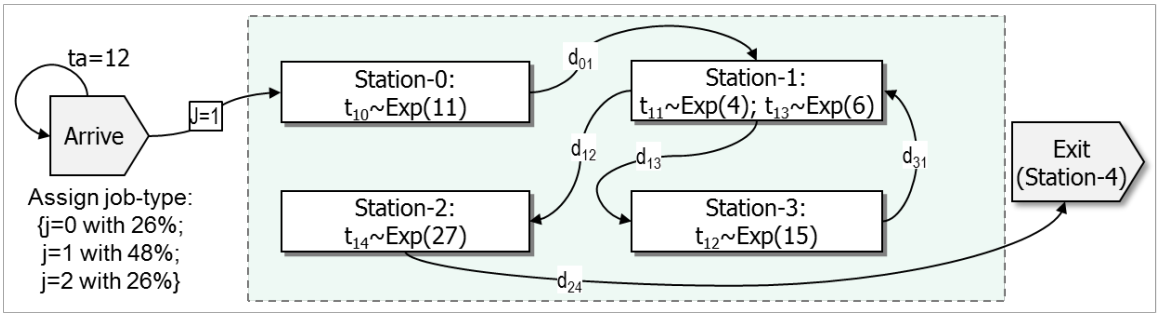

Fig. 1. Routing sequence and processing times of the type 1 job (j = 1)

Table 2. Transport delay data

delay[v, w]= d_{vw}		to Station (w)				
		0	1	2	3	4
from Station (v)	0	0	d_{01}=2	d_{02}=4	d_{03}=6	d_{04}=2
	1	d_{10}=6	0	d_{12}=2	d_{13}=4	d_{14}=2
	2	d_{20}=4	d_{21}=6	0	d_{23}=2	d_{24}=2
	3	d_{30}=2	d_{31}=4	d_{32}=6	0	d_{34}=2

With the input data of the simple job shop as given in Tables 1 and 2, we're going to collect the *average queue length* (AQL) statistics for each station during the simulation.

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

Figure 2 reproduces the parameterized event graph (PEG) model of the simple job shop given in Section 5.5.2 of Chapter 5 (Fig. 5.18) of the textbook, where t_a (=12) and t_s (=30) denote the *inter-arrival time* and ***setup time***, respectively. The *processing time* is denoted by t_p. The *routing sequence* data, *processing time* data, and *transport delay* data are read in as arrays route[j,k], t[j,k], and delay[s, ns], respectively.

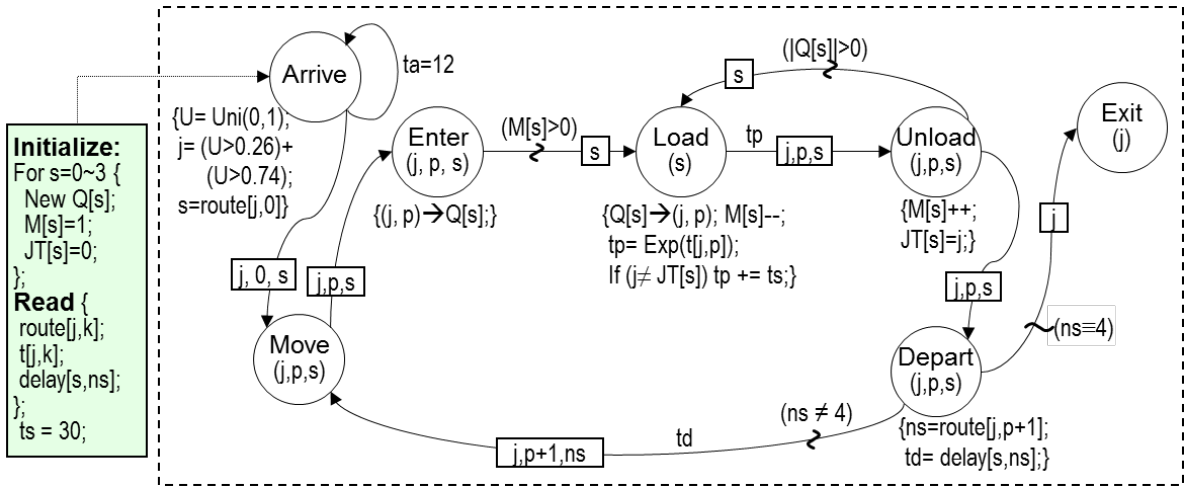

Fig. 2. PEG Model of the Simple Job Shop

Table 3 shows the variables used in the simple job shop PEG model: (1) state variables *j, p, s,* and *ns*; (2) time variables *tp, td, t[j,p],* and *delay[s, ns]*; (3) route variable *route[j,p]*.

Table 3. Variables Declared for the Simple Job Shop PEG Model in Fig. 2

Category	Name	Description
State Variable	j	job type of a job
	p	current processing step of a job
	s	current station number for a job
	ns	next station number for a job
Time Variable	tp	processing time for a job including a setup time if necessary
	td	transport delay for a job
	t[j, p]	the processing time for a job having job-type j and processing-step p
	delay[s, ns]	transport delay from station s to station ns
Route Variable	route[j, p]	the next station for a job having job-type j and processing-step p

Table 4 shows the *event transition table* for the event graph model shown in Fig. 2. Notice that the Initialize box in the PEG model is treated as an event. Listed in the event transition table are eight event types: Initialize, Arrive, Move(j,p,s), Enter(j,p,s), Load(s), Unload(j,p,s), Depart(j,p,s), and Exit(j).

Table 4. Event Transition Table of the Simple Job Shop PEG Model in Fig. 2

No	Event	State Change	Edge	Cond.	Delay	Parameter	Next Event
0	Initialize	For s=0~3 {New Q[s]; M[s]=1; JT[s]= 0 }; Read { route[j, k], t[j, k]} for j=0~2 and k=0~5; Read { delay[j, k]} for j=0~3 and k=0~4; ts = 30;	1	True	0		Arrive
1	Arrive	U=Uni(0,1); j=(U>0.26)+(U>0.74); s =route[j,0];	1	True	12	-	Arrive
			2	True	0	j,0,s	Move(j,p,s)
2	Move(j,p,s)		1	True	0	j,p,s	Enter(j,p,s)
3	Enter(j,p,s)	(j, p)→Q[s];	1	M[S]>0	0	s	Load(s)
4	Load(s)	Q[s]→(j, p); M[s]--; tp= Exp(t[j, p]); If (j≠ JT[s]) tp += ts;	1	True	tp	j,p,s	Unload(j,p,s)
5	Unload(j,p,s)	M[s]++; JT[s]=j;	1	True	0	j,p,s	Depart(j,p,s)
			2	|Q[s]|>0	0	s	Load(s)
6	Depart(j,p,s)	ns=route[j, p+1]; td= delay[s, ns];	1	ns != 4	td	j,p+1,ns	Move(j,p,s)
			2	ns ≡ 4	0	j	Exit(j)
7	Exit(j)		1				

1.2 Augmented Event Transition Table for Collecting Statistics

In order to collect the AQL statistics, the following *statistics variables* are introduced: (1) SumQ[s] = sum of queue length changes over time at station *s*, (2) Bef[s] = previous queue length change time of *Q[k]* at station *s*, and (3) AQL[s] = average queue length for each station *s*. And then, the *event transition table* (ETT) is augmented as follows:

① SumQ[s] and Bef[s] are initialized at Initialize event of the ETT:
- For s=0~3 {SumQ[s] = Bef[s] = 0;}

② SumQ[s] and Bef[s] are updated at the state changes of Enter(j,p,s) and Load(s) events:
- SumQ[s] += |Q[s]|*(CLK – Bef[s]); Bef[s] = CLK;

③ ***Statistics event*** is newly introduced where AQL[s] is computed as follows:
- For s=0~3 {SumQ[s] += |Q[s]| * (CLK – Bef[s]); AQL[s] = SumQ[s] / CLK;}

Table 5. Augmented Event Transition Table for collecting the average queue lengths

No	Event	State Change	Edge	Cond.	Delay	Parameter	Next Event
0	Initialize	For s=0~3 {New Q[s]; M[s]=1; JT[s]=SumQ[s]=Bef[s]= 0 }; Read { route[j, k], t[j, k]} for j=0~2 and k=0~5; Read { delay[j, k]} for j=0~3 and k=0~4; ts = 30;	1	True	0		Arrive
1	Arrive	U=Uni(0,1); j=(U>0.26)+(U>0.74); s =route[j,0];	1	True	12	-	Arrive
			2	True	0	j,0,s	Move(j,p,s)
2	Move(j,p,s)		1	True	0	j,p,s	Enter(j,p,s)
3	Enter(j,p,s)	SumQ[s] +=|Q[s]|*(CLK–Bef[s]); Bef[s]=CLK; (j, p)→Q[s];	1	M[S]>0	0	s	Load(s)
4	Load(s)	SumQ[s] += |Q[s]|*(CLK–Bef[s]); Bef[s] = CLK; Q[s]→(j, p); M[s]--; tp= Exp(t[j, p]); If (j≠ JT[s]) tp += ts;	1	True	tp	j,p,s	Unload(j,p,s)
5	Unload(j,p,s)	M[s]++; JT[s]=j;	1	True	0	j,p,s	Depart(j,p,s)
			2	|Q[s]|>0	0	s	Load(s)
6	Depart(j,p,s)	ns=route[j, p+1]; td= delay[s, ns];	1	ns != 4	td	j,p+1,ns	Move(j,p,s)
			2	ns ≡ 4	0	j	Exit(j)
7	Exit(j)		1				
8	Statistics	For s=0~3 { SumQ[s] += |Q(s)|*(CLK-Bef[s]); AQL[s] = SumQ[s] / CLK;}					

2. Developing a Dedicated PEG Simulator

This section describes how a dedicated PEG simulator for a simple job shop is developed. The 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 ***"simplejobshopsimulator.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 "***Simple Job Shop PEG Simulator.sln***".

2.2 Source Code Structure and Class Diagram

The project, named "*Simple Job Shop PEG Simulator*", contains the following files:

- `Simulator.cs`: *Simulator* class that has 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
- `MainFrm.cs`: Graphical user interface for the simulator
- `Program.cs`: entry point of the program (do not modify this code)

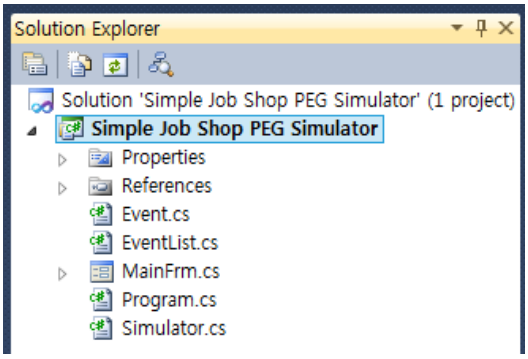

Fig. 3. Source Code Structure shown in Solution Explorer of Visual Studio 2010

Figure 4 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, seven ***event routines*** (*Arrive, Move, Enter, Load, Unload, Depart,* and *Exit*), ***list-handling methods*** (*Schedule,* and *Retrieve*) and ***random variate generators*** (*Exp and Uni*).

[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

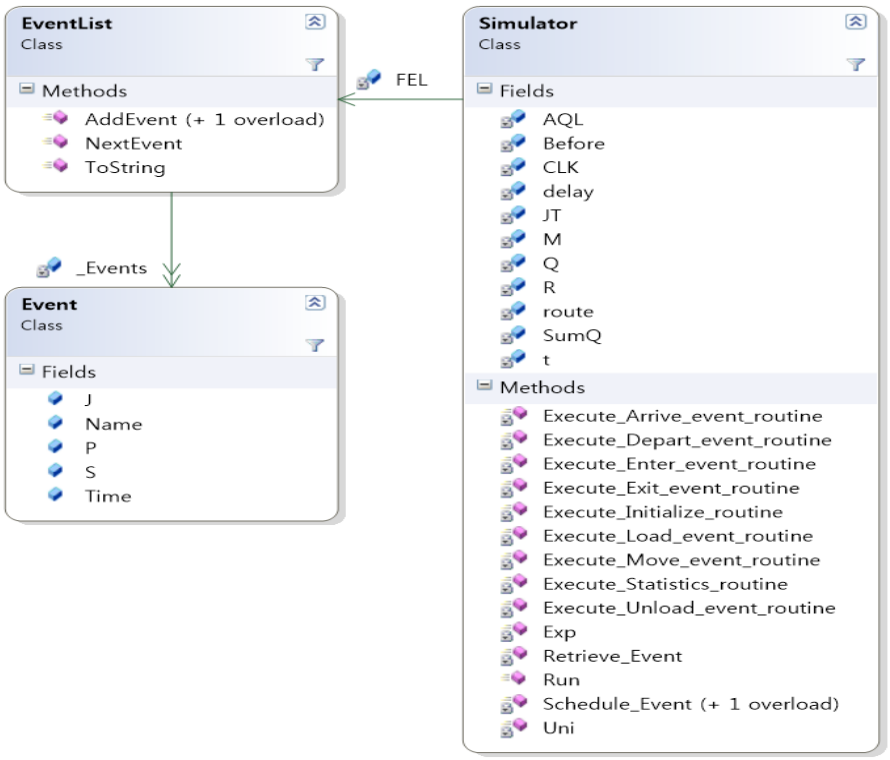

Fig. 4. Class Diagram of the Dedicated Simulator

The ***member variables*** in the *Simulator* class include: (1) *state variables M[], Q[], JT[]*; (2) simulation *clock CLK*; (3) *statistics variables SumQ[], Before[]* and *AQL[]*; (4) *time variables t[]* and *delay[]*; (5) route variable *route[]*; (6) *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 five *properties*: *J* (job), *P* (processing step), *S* (station), *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 phase, (2) Time-flow mechanism phase, (3) Event-routine execution phase, and (4) Statistics calculation phase.

```
public void Run(double eosTime) {
    //1. Initialization phase
    FEL = new EventList();
    R = new Random();
    Event nextEvent = new Event();
    CLK = 0.0;

    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(
                  nextEvent.J, nextEvent.P, nextEvent.S, CLK); break; }
            case "Move": {
                Execute_Move_event_routine(
                  nextEvent.J, nextEvent.P, nextEvent.S, CLK); break; }
            case "Enter": {
                Execute_Enter_event_routine(
                  nextEvent.J, nextEvent.P,nextEvent.S,CLK); break; }
            case "Load":{
                Execute_Load_event_routine(
                  nextEvent.J, nextEvent.P, nextEvent.S, CLK); break; }
            case "Unload":{
                Execute_Unload_event_routine(
                  nextEvent.J, nextEvent.P, nextEvent.S, CLK);break;}
            case "Depart": {
                Execute_Depart_event_routine(
                  nextEvent.J, nextEvent.P, nextEvent.S, CLK); break; }
            case "Exit":{
                Execute_Exit_event_routine(
                  nextEvent.J, nextEvent.P, nextEvent.S, 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[] = JT[] = 0; M[] = 1), statistics variables (Before[] = 0; SumQ[] = 0), reads route and time variables (route[], t[], delay[]), and schedules the ***initial event*** *"Arrive"* by invoking *Schedule_Event ("Arrive", Now)*.

```
private void Execute_Initialize_routine(double Now) {
    Q = new Queue<int[]>[4]; M = new int[4]; JT = new int[4];
    Before = new double[4]; SumQ = new double[4];
    for (int s = 0; s < 4; s++) {
        Q[s] = new Queue<int[]>();
        M[s] = 1;
        JT[s] = 0;
        Before[s] = 0.0;
        SumQ[s] = 0.0;
    }

    //Initialize the routes
    route = new int[3, 6];
    route[0,0]=0; route[0,1]=1; route[0,2]=2; route[0,3]=3; route[0,4]=4;
    route[1,0]=0; route[1,1]=1; route[1,2]=3; route[1,3]=1; route[1,4]=2;
    route[1,5]=4;
```

```
    route[2,0]=1; route[2,1]=0; route[2,2]=2; route[2,3]=4;

    //Initialize the processing times
    t = new double[3, 6];
    t[0, 0] = 6 ;  t[0, 1] = 5; t[0, 2] = 15; t[0, 3] = 8;
    t[1, 0] = 11;  t[1, 1] = 4; t[1, 2] = 15; t[1, 3] = 6; t[1, 4] = 27;
    t[2, 0] = 7 ;  t[2, 1] = 7; t[2, 2] = 18;

    //Initialize the transport delay data
    delay = new int[4, 5];
    delay[0,0]=0; delay[0,1]=2; delay[0,2]=4; delay[0,3]=6; delay[0,4]=2;
    delay[1,0]=6; delay[1,1]=0; delay[1,2]=2; delay[1,3]=4; delay[1,4]=2;
    delay[2,0]=4; delay[2,1]=6; delay[2,2]=0; delay[2,3]=2; delay[2,4]=2;
    delay[3,0]=2; delay[3,1]=4; delay[3,2]=6; delay[3,3]=0; delay[3,4]=2;

    //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, AQL[s] (average queue lengths for s= 0 ~ 3) are obtained by invoking Execute_Statistics_routine which is defined as below:

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

2.4 Event Routines

In the PEG model of the simple job shop (Fig. 2), the parameter variables are not equal among the events, which may cause difficulty in implementing the event routines. A simple method to avoid this difficulty is to use the same-size list for all event routines. In this document, the *parameter list* (j, p, s) is used for all event routines. Therefore, the *event-routine methods* in the Simulator class are defined as follows:

(a) `Execute_Arrive_event_routine (j, p, s, Now)`,
(b) `Execute_Move_event_routine (j, p, s, Now)`,
(c) `Execute_Enter_event_routine (j, p, s, Now)`,
(d) `Execute_Load_event_routine (j, p, s, Now)`,
(e) `Execute_Unload_event_routine (j, p, s, Now)`,
(f) `Execute_Depart_event_routine (j, p, s, Now)`, and
(g) `Execute_Exit_event_routine (j, p, s, Now)`.

An event routine is a subprogram describing the changes in state variables and how the next *destination events* are scheduled and/or canceled for an *originating event*. One event routine is required for each event node 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 *Schedule_Event ()*.

```
private void Execute_Arrive_event_routine(int j, int p, int s, double Now)
{
    double U = Uni(0, 1);
    j = ((U > 0.26) ? 1 : 0) + ((U > 0.74) ? 1 : 0); // assign job type j
    s = route[j, 0];

    Schedule_Event("Arrive", 0, 0, 0, Now + 12);
    Schedule_Event("Move", j, 0, s, Now);
}
```

```
private void Execute_Move_event_routine(int j, int p, int s, double Now)
{
    //Schedule Event
    Schedule_Event("Enter", j, p, s, Now);
}
```

```
private void Execute_Enter_event_routine(int j, int p, int s, double Now)
{
    //Collect Statistics
    SumQ[s] += Q[s].Count * (Now - Before[s]); Before[s] = Now;

    //State Change
    Q[s].Enqueue(new int[] { j, p });

    //Schedule Event
    if (M[s] > 0)
        Schedule_Event("Load", 0, 0, s, Now);
}
```

```
private void Execute_Load_event_routine(int j, int p, int s, double Now)
{
    //Collect Statistics
    SumQ[s] += Q[s].Count * (Now - Before[s]); Before[s] = Now;

    //State change
    int[] job = Q[s].Dequeue();
    j = job[0]; p = job[1];
    M[s]--;

    double tp = Exp(t[j, p]);
    if (j != JT[s]) tp += 30;

    //Schedule Event
    Schedule_Event("Unload", j, p, s, Now + tp);
}
```

```
private void Execute_Unload_event_routine(int j, int p, int s, double Now)
{
    //State change
    M[s]++;
    JT[s] = j;

    //Schedule Event
    if (true)
```

```
        Schedule_Event("Depart", j, p, s, Now);
    if (Q[s].Count > 0)
        Schedule_Event("Load", 0, 0, s, Now);
}
```

```
private void Execute_Depart_event_routine(int j, int p, int s, double Now)
{
    int ns = route[j, p + 1];
    int td = delay[s, ns];

    if (ns != 4)
        Schedule_Event("Move", j, p + 1, ns, Now + td);
    else if (ns == 4)
        Schedule_Event("Exit", j, 0, 0, Now);
}
```

```
private void Execute_Exit_event_routine(int j, int p, int s, double Now)
{
    //Do Nothing
}
```

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` and `Retrieve_Event` methods. `Schedule_Event` is invoked at the *event routines* and `Retrieve_Event` is invoked at the *time-flow mechanism* phase of the main program. Here, the `Schedule_Event` method is overloaded with different method signatures: one with name and time only and the other with additional arguments j, p, s. The list-handling methods are defined as follows:

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

private void Schedule_Event(string name, int j, int p, int s, double time)
{
    FEL.AddEvent(name, j, p, s, 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*. 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);
}
```

The 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 the 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, "SimpleJobShopSimulator.exe" under a folder of "SimpleJobShopSimulator\bin\Debug".

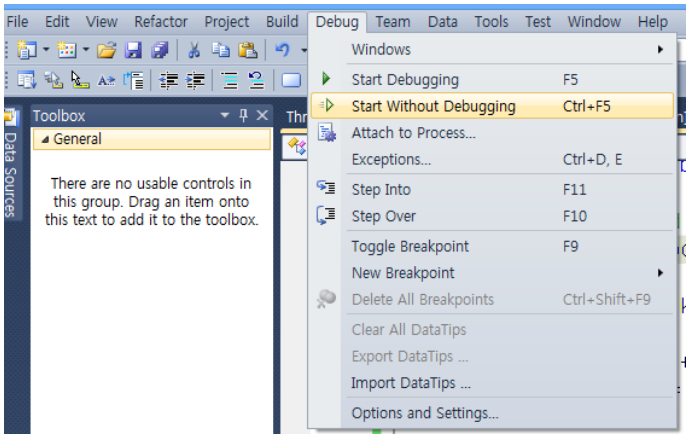

Fig. 5. Run the Dedicated Simulator from Visual Studio

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):

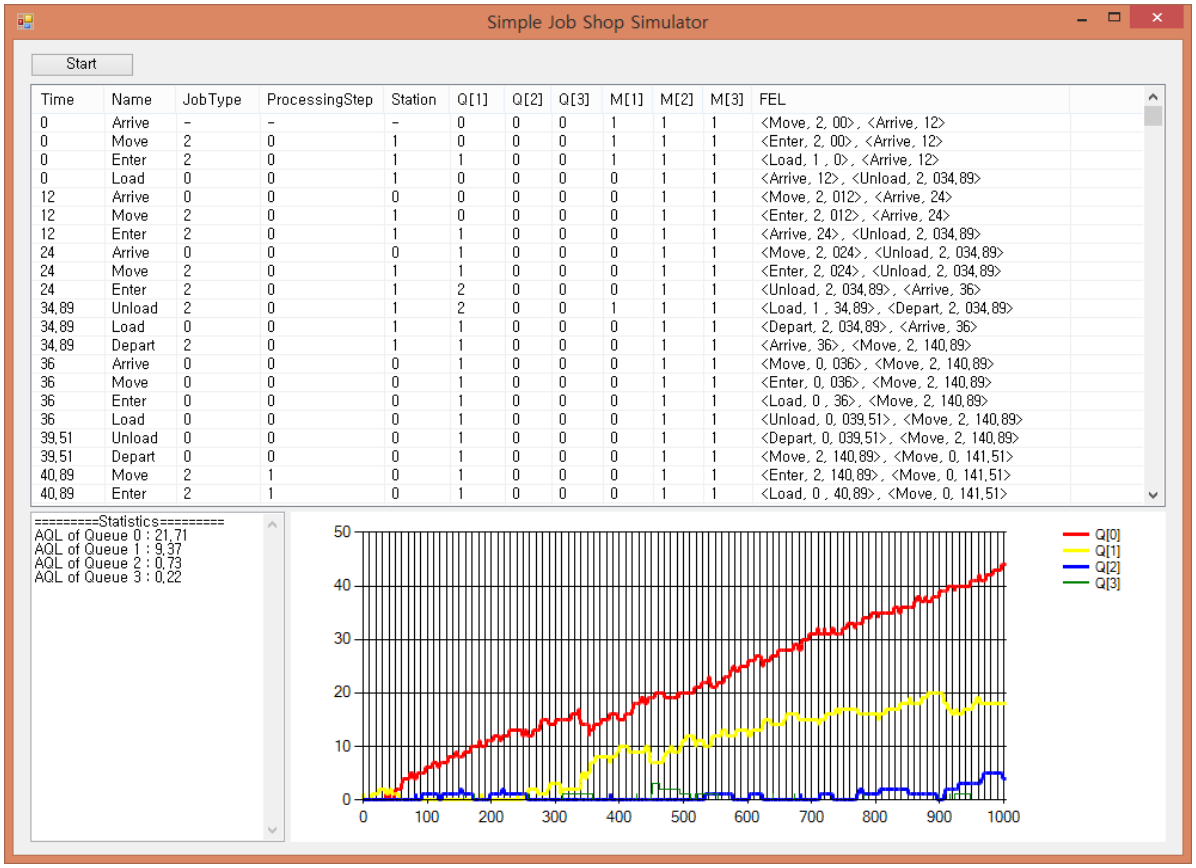

Fig. 6. Dedicated Simulator with system trajectory and AQL statistics

In the System Trajectory, you can observe how the system states change over time. First five columns are Time, Name, JobType, ProcessingStep, and Station where an event (Name) is occurred at a Time with parameters (JobType, ProcessingStep, Station). The following six columns represent the values of *Q[s]* and *M[s]* for s that varies from 0 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. 6 are AQLs for the four queues Q[s] for s = 0 ~ 3. Displayed right to the AQL part are queue-length graphs for the three queues.

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.SimpleJobShop {
    /// <summary>
    /// Class for an Event Record
    /// </summary>
    public class Event {
        /// <summary>
        /// Name of Event
        /// </summary>
        public string Name;
        /// <summary>
        /// Scheduled Time of Event
        /// </summary>
        public double Time;
        /// <summary>
        /// Job Type (Event Parameter)
        /// </summary>
        public int J;
        /// <summary>
        /// Processing Step (Event Parameter)
        /// </summary>
        public int P;
        /// <summary>
        /// Station Number (Event Parameter)
        /// </summary>
        public int S;

        /// <summary>
        /// Parameterless default constructor
        /// </summary>
        public Event() {
            this.J = 0;
            this.P = 0;
            this.S = 0;
        }

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="name"></param>
        /// <param name="time"></param>
        public Event(string name, double time) : this() {
            this.Name = name;
            this.Time = time;
        }

        /// <summary>
        /// Constructor with parameters
        /// </summary>
        /// <param name="name"></param>
        /// <param name="j"></param>
        /// <param name="p"></param>
        /// <param name="s"></param>
        /// <param name="time"></param>
        public Event(string name, int j, int p, int s, double time) {
            this.Name = name;
            this.P = p;
            this.J = j;
            this.S = s;
```

```
            this.Time = time;
        }

        /// <summary>
        /// Check if this event is equal to the parameter
        /// </summary>
        /// <param name="obj">Object to compare with this event</param>
        /// <returns>true, if this event is equal to the parameter,
        ///          false, otherwise.</returns>
        public override bool Equals(object obj) {
            if (obj is Event) {
                Event target = obj as Event;
                if (target.Name == this.Name && target.J == this.J &&
                    target.P == this.P && target.S == this.S &&
                    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.SimpleJobShop {
    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();
            nextEvent.Name = eventName;
            nextEvent.Time = 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);
            }
        }

        /// <summary>
        /// Schedule an event into the future event list (FEL)
        /// </summary>
        /// <param name="name">the name of event</param>
        /// <param name="j">the job type of event</param>
        /// <param name="p">the processing step of event</param>
        /// <param name="s">the station of event</param>
        /// <param name="time">the time of event</param>
        public void AddEvent(string name, int j, int p, int s, double
time) {
            Event nextEvent = new Event();
            nextEvent.Name = name;
            nextEvent.J = j;
            nextEvent.P = p;
            nextEvent.S = s;
            nextEvent.Time = time;
            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
        /// </summary>
        public void RemoveEvent(string name, int j, int p, int s)
        {
            Event CancelEvent = null;
            for (int i = 0; i < _Events.Count; i++) {
                Event e = _Events[i];
                if (e.Name == name && e.J == j && e.P == p && e.S == s) {
                    CancelEvent = e; break;
                }
            }
            if (CancelEvent != null)
                _Events.Remove(CancelEvent);
        }

        /// <summary>
        /// Return the contents of FEL
        /// </summary>
        /// <returns>the name and time of 1st and 2nd FEL</returns>
        public override string ToString() {
            string fel = "";

            int num;
            if (_Events.Count < 2)
                num = _Events.Count;
            else
                num = 2;

            for (int i = 0; i < num; i++) {
                double fel_time = Math.Round(_Events[i].Time, 2);
                if (i != 0)
                    fel += ", ";
                if (_Events[i].Name == "Arrive")
                    fel += "<" + _Events[i].Name + ", " + fel_time + ">";
                else if (_Events[i].Name == "Exit")
                    fel += "<" + _Events[i].Name + ", " + _Events[i].J + ",
" + fel_time + ">";
                else if (_Events[i].Name == "Load")
                    fel += "<" + _Events[i].Name + ", " + _Events[i].S + " ,
" + fel_time + ">";
                else
                    fel += "<" + _Events[i].Name + ", " +
_Events[i].J.ToString() + ", " + _Events[i].P.ToString() + fel_time +
">";
            }
            return fel;
        }
        #endregion
    }
}
```

4.3 Simulator.cs

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

```
namespace MSDES.Chap05.SimpleJobShop {
    public class Simulator {
        #region Member Variables for State Variables
        /// <summary>
        /// Number of Available Machines at each station
        /// </summary>
        private int[] M;
        /// <summary>
        /// Queue (List of job(j,p)) at each station
        /// </summary>
        private Queue<int[]>[] Q;
        /// <summary>
        /// Current Job Type at each station
        /// </summary>
        private int[] JT;
        /// <summary>
        /// Routes of each job type
        /// </summary>
        private int[,] route;
        /// <summary>
        /// Procesing Time for a job having job type k and processing-step
p
        /// </summary>
        private double[,] t;
        /// <summary>
        /// Transport Delay from Station s to Station ns
        /// </summary>
        private int[,] delay;
        #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
        /// <summary>
        /// Lastly Collected Times
        /// </summary>
        private double[] Before;
        /// <summary>
        /// Accumulated Values of Queue Length Changes over Time
        /// </summary>
        private double[] SumQ;
        /// <summary>
        /// Average Queue Lengths
        /// </summary>
        private double[] AQL;
        #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
            FEL = new EventList();
            R = new Random();
            Event nextEvent = new Event();
            CLK = 0.0;

            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(
                         nextEvent.J, nextEvent.P, nextEvent.S, CLK); break;
                    }
                    case "Move": {
                        Execute_Move_event_routine(
                         nextEvent.J, nextEvent.P, nextEvent.S, CLK); break; }
                    case "Enter": {
                        Execute_Enter_event_routine(
                         nextEvent.J, nextEvent.P, nextEvent.S, CLK); break;}
                    case "Load":{
                        Execute_Load_event_routine(
                         nextEvent.J, nextEvent.P, nextEvent.S, CLK); break;}
                    case "Unload":{
                        Execute_Unload_event_routine(
                         nextEvent.J, nextEvent.P, nextEvent.S, CLK);break;}
                    case "Depart": {
                        Execute_Depart_event_routine(
                         nextEvent.J, nextEvent.P, nextEvent.S, CLK); break;}
                    case "Exit":{
                        Execute_Exit_event_routine(
                         nextEvent.J, nextEvent.P, nextEvent.S, CLK); break;}
                }

                //Print out the event trajectory and the graph of Queues
```

```
                MainFrm.App.AddTrajectory(Math.Round(nextEvent.Time, 2),
nextEvent.Name, nextEvent.J, nextEvent.P, nextEvent.S, Q, M,
FEL.ToString());
                MainFrm.App.AddChart(CLK, Q[0].Count, Q[1].Count,
Q[2].Count, Q[3].Count);
            }
            //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="time">Event Time</param>
        private void Schedule_Event(string name, double time)  {
            FEL.AddEvent(name, time);
        }

        /// <summary>
        /// Schedule an event into the future event list (FEL)
        /// </summary>
        /// <param name="name">the name of event</param>
        /// <param name="j">the job type of event</param>
        /// <param name="p">the processing step of event</param>
        /// <param name="s">the station of event</param>
        /// <param name="time">the time of event</param>
        private void Schedule_Event(string name, int j, int p, int s,
double time) {
            FEL.AddEvent(name, j, p, s, time);
        }

        /// <summary>
        /// Return an event record that located at the first element in the
future event list(FEL).
        /// When there is no more event record in the FEL, this method
returns null.
        /// </summary>
        /// <returns>An event record</returns>
        private Event Retrieve_Event()
        {
            Event nextEvent = null;
            nextEvent = FEL.NextEvent();
            return nextEvent;
        }
        #endregion

        #region Event Routines
        /// <summary>
        /// Execute initialize routine
        /// </summary>
        /// <param name="Now">Time</param>
        private void Execute_Initialize_routine(double Now)
        {
            Q = new Queue<int[]>[4]; M = new int[4]; JT = new int[4];
            Before = new double[4]; SumQ = new double[4];
            for (int s = 0; s < 4; s++)
            {
                Q[s] = new Queue<int[]>();
                M[s] = 1;
```

```
            JT[s] = 0;
            Before[s] = 0.0; SumQ[s] = 0.0;
        }

        //Initialize the routes
        route = new int[3, 6];
        route[0, 0] = 0; route[0, 1] = 1; route[0, 2] = 2;
        route[0, 3] = 3; route[0, 4] = 4;
        route[1, 0] = 0; route[1, 1] = 1; route[1, 2] = 3;
        route[1, 3] = 1; route[1, 4] = 2; route[1, 5] = 4;
        route[2, 0] = 1; route[2, 1] = 0; route[2, 2] = 2;
        route[2, 3] = 4;

        //Initialize the processing times
        t = new double[3, 6];
        t[0, 0] = 6; t[0, 1] = 5; t[0, 2] = 15; t[0, 3] = 8;
        t[1, 0] = 11; t[1, 1] = 4; t[1, 2] = 15; t[1, 3] = 6;
        t[1, 4] = 27;
        t[2, 0] = 7; t[2, 1] = 7; t[2, 2] = 18;

        //Initialize the transport delay data
        delay = new int[4, 5];
        delay[0, 0] = 0; delay[0, 1] = 2; delay[0, 2] = 4;
        delay[0, 3] = 6; delay[0, 4] = 2;
        delay[1, 0] = 6; delay[1, 1] = 0; delay[1, 2] = 2;
        delay[1, 3] = 4; delay[1, 4] = 2;
        delay[2, 0] = 4; delay[2, 1] = 6; delay[2, 2] = 0;
        delay[2, 3] = 2; delay[2, 4] = 2;
        delay[3, 0] = 2; delay[3, 1] = 4; delay[3, 2] = 6;
        delay[3, 3] = 0; delay[3, 4] = 2;

        //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(int j, int p, int s,
double Now)  {
        double U = Uni(0, 1);
        j = ((U > 0.26) ? 1 : 0) + ((U > 0.74) ? 1 : 0);
        s = route[j, 0];

        Schedule_Event("Arrive", 0, 0, 0, Now + 12);
        Schedule_Event("Move", j, 0, s, Now);
    }

    /// <summary>
    /// Execute Move event routine
    /// </summary>
    /// <param name="j">JobType</param>
    /// <param name="p">ProcessingStep</param>
    /// <param name="s">Station</param>
    /// <param name="Now">Current Simulation Clock</param>
    private void Execute_Move_event_routine(int j, int p, int s, double
Now) {
        //Schedule Event
        Schedule_Event("Enter", j, p, s, Now);
    }
```

```
        /// <summary>
        /// Execute Enter event routine
        /// </summary>
        /// <param name="j">JobType</param>
        /// <param name="p">ProcessingStep</param>
        /// <param name="s">Station</param>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Enter_event_routine(int j, int p, int s,
double Now) {
            //Collect Statistics
            SumQ[s] += Q[s].Count * (Now - Before[s]); Before[s] = Now;

            //State Change
            Q[s].Enqueue(new int[] { j, p });

            //Schedule Event
            if (M[s] > 0)
                Schedule_Event("Load", 0, 0, s, Now);
        }

        /// <summary>
        /// Execute Load event routine
        /// </summary>
        /// <param name="s">Station</param>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Load_event_routine(int j, int p, int s, double
Now) {
            //Collect Statistics
            SumQ[s] += Q[s].Count * (Now - Before[s]); Before[s] = Now;

            //State change
            int[] job = Q[s].Dequeue();
            j = job[0];
            p = job[1];
            M[s]--;

            double tp = Exp(t[j, p]);
            if (j != JT[s]) tp += 30;
            //Schedule Event
            Schedule_Event("Unload", j, p, s, Now + tp);
        }

        /// <summary>
        /// Execute Unload event routine
        /// </summary>
        /// <param name="j">JobType</param>
        /// <param name="p">ProcessingStep</param>
        /// <param name="s">Station</param>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Unload_event_routine(int j, int p, int s,
double Now) {
            //State change
            M[s]++;
            JT[s] = j;

            //Schedule Event
            if (true)
                Schedule_Event("Depart", j, p, s, Now);
            if (Q[s].Count > 0)
                Schedule_Event("Load", 0, 0, s, Now);
        }
```

```
        /// <summary>
        /// Execute Depart event routine
        /// </summary>
        /// <param name="j">JobType</param>
        /// <param name="p">ProcessingStep</param>
        /// <param name="s">Station</param>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Depart_event_routine(int j, int p, int s,
double Now) {
            int ns = route[j, p + 1];
            int td = delay[s, ns];

            if (ns != 4)
                Schedule_Event("Move", j, p + 1, ns, Now + td);
            else if (ns == 4)
                Schedule_Event("Exit", j, 0, 0, Now);
        }
        /// <summary>
        /// Execute Exit event routine
        /// </summary>
        /// <param name="j">JobType</param>
        /// <param name="Now">Current Simulation Clock</param>
        private void Execute_Exit_event_routine(int j, int p, int s, double
Now)
        {
            //Do Nothing
        }

        /// <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) {
            AQL = new double[4];
            for (int s = 0; s < 4; s++) {
                SumQ[s] += Q[s].Count * (Now - Before[s]);
                AQL[s] = SumQ[s] / Now;
            }
        }
        #endregion

        #region Random-variate Generation Methods
        /// <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>Random value of the exponential distribution</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>
        /// 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
    }
}
```