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

November, 2013

Byoung K. Choi and Donghun Kang

Objective

This document provides a guide to the *three-stage tandem line P-ACD simulator* presented in Section 10.2.4 of the textbook, *Modeling and Simulation of Discrete-Event Systems*. It gives a technical description of how the dedicated P-ACD simulator is implemented in C# language.

Recommendation

Prior to reading this document, the readers are recommended to read and understand Section 10.2.4 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**
11/18/2013	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

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 *three-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 that the distributions of the inter-arrival times 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 AQL (*average queue length*) statistics during the simulation.

1.1 Parameterized Activity Cycle Diagram (P-ACD) Model

Figure 1 shows the P-ACD model of the three-stage tandem line introduced in the textbook (See Fig. 10.8 in Section 10.2.4 of Chapter 10), where B[k] and M[k] denote the numbers of jobs in the buffer and available machines at stage k, respectively. Table 1 shows the activity transition table of the P-ACD model in Fig. 1.

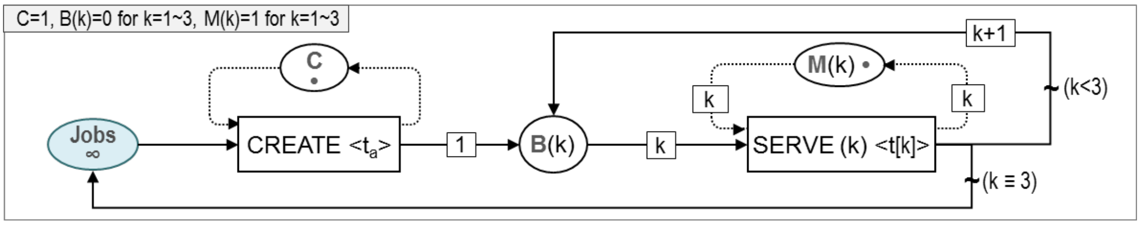


Fig. 1. P-ACD Model of the Three-stage Tandem Line

Table 1. Activity Transition Table of the Three-stage Tandem Line Model in Fig. 1

No	Activity	At-begin		BTO-event		At-end				
		Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Infl. Act.
1	CREATE	(C>0)	C--;	t_a	CREATED	1	True	-	C++;	CREATE
						2	True	1	B[k]++;	SERVE(k)
2	SERVE(k)	(B[k]>0) & (M[k]>0)	B[k]--; M[k]--;	t[k]	SERVED	1	True	k	M[k]++;	SERVE(k)
						2	(k < 3)	k+1	B[k]++;	SERVE(k)
						3	(k ≡ 3)	-	-	-
Initialize		*Initial Marking* = {C=1, B[k]=0 for k=1~3, M[k]=1 for k=1~3}; *Enabled Activities* = {CREATE}								

1.2 Augmented Activity Transition Table for Collecting Statistics

In order to collect the AQL statistics, the following *statistics variables* are introduced: (1) SumQ[k] = sum of queue lengths B[k], (2) Before[k] = previous change time of B[k], and (3) AQL[k] = average queue length at stage k. And then, the *activity transition table* (ATT) is augmented as follows:

① SumQ[k] and Before[k] are initialized at the Initialization entry of the ATT:

- SumQ[k] = Before[k] = 0 for k = 1 ~ 3

② SumQ[k] and Before[k] are updated at the (1) At-end Action entry of CREATE, (2) At-begin Action entry of SERVER(k), and (3) At-end Action entry of SERVER(k) of the ATT:

- SumQ[k] += B[k] * (Clock – Before[k]); Before[k] = Clock;

③ SumQ[k] and AQL[k] are computed at the Statistics entry of the ATT:

- { SumQ[k] += B[k] * (Clock – Before[k]); AQL[k] = SumQ[k] / Clock; } for k = 1 ~ 3

Thus, the augmented activity transition table is obtained as in Table 2 which will be used in developing the dedicated simulator.

Table 2. Augmented Activity Transition Table for collecting the average queue lengths

No	Activity	At-begin		BTO-event		At-end				
		Condition	Action	Time	Name	Arc	Condition	Parameter	Action	Infl. Act.
1	CREATE	(C>0)	C--;	t_a	CREATED	1	True	-	C++;	CREATE
						2	True	1	SumQ[k] += B[k] * (Clock - Before[k]); Before[k] = Clock; B[k]++;	SERVE(k)
2	SERVE(k)	(B[k]>0) && (M[k]>0)	SumQ[k] += B[k] * (Clock - Before[k]); Before[k] = Clock; B[k]--; M[k]--;	t[k]	SERVED	1	True	k	M[k]++;	SERVE(k)
						2	(k < 3)	k+1	SumQ[k] += B[k] * (Clock - Before[k]); Before[k] = Clock; B[k]++;	SERVE(k)
						3	(k ≡ 3)	-	-	-
Initialize		*Initial Marking* = {C=1, B[k]=0 and M[k]=1 for k = 1 ~ 3}; *Enabled Activities* = {CREATE} *Variables* = {SumQ[k]=Before[k]=0 for k = 1 ~ 3};								
Statistics		{ SumQ[k] += B[k] * (Clock – Before[k]); AQL[k] = SumQ[k] / Clock; } for k = 1 ~ 3								

2. Developing a Dedicated P-ACD Simulator for 3-stage tandem line

This section describes how a dedicated P-ACD simulator for the three-stage tandem line is developed. C# codes are based on the pseudo codes given in Section 10.2.4 of the textbook.

2.1 Development Environment

The dedicated simulator was developed with Microsoft Visual Studio 2010 and compiled with Microsoft .NET Framework Version 4.0. If you have Microsoft Visual Studio 2010[1], please unzip the ***"threestagetandemlinesimulator.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/acdsimulator), into a folder an d open the solution file, which is named "***Three-stage Tandem Line Simulator.sln***".

[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

2.2 Source Code Structure and Class Diagram

The project, named "*Three-stage Tandem Line Simulator*", contains the following files:

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

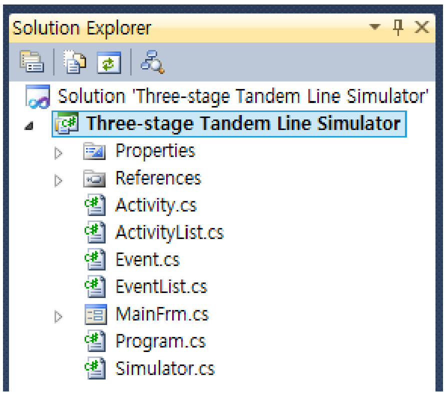


Fig. 2. Source Code Structure shown in Solution Explorer of Visual Studio 2010

Figure 3 shows the class diagram consisting of five classes: *Simulator*, *EventList*, *Event*, *ActivityList*, and *Activity* classes. The *Simulator class* contains ***Main program*** (*Run*) together with ***activity routines*** (*CREATE* and *SERVE*), ***event routines*** (*CREATED* and *SERVED*), ***list-handling methods*** (*Store_Activity*, *Get_Activity*, *Schedule_Event*, and *Retrieve_Event*), and ***random variate generators*** (*Exp and Uni*).

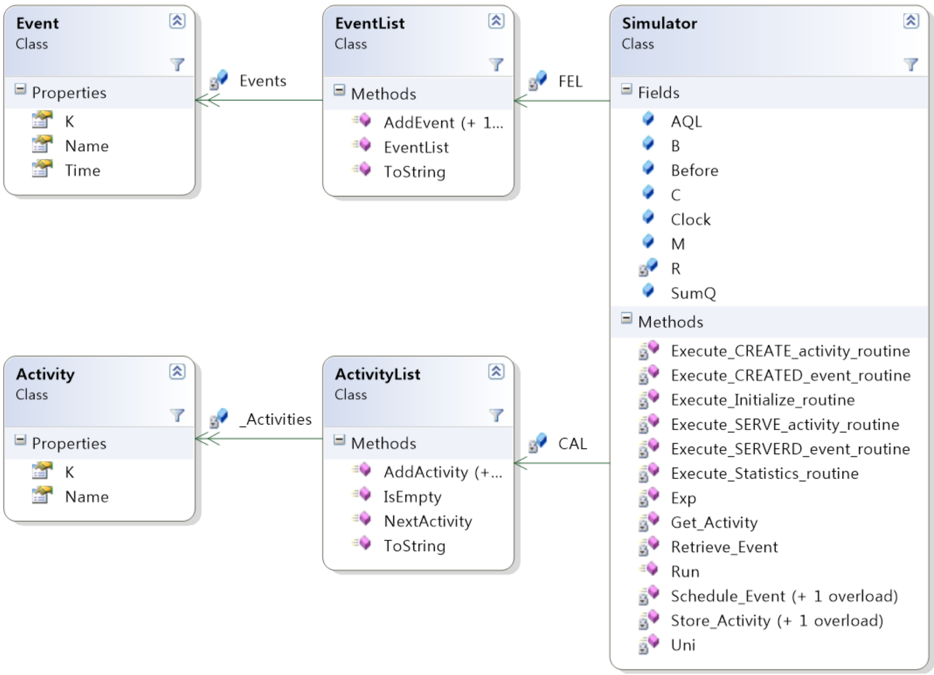


Fig. 3. Class Diagram of the Dedicated Simulator

The ***member variables*** in the *Simulator* class include: (1) *state variables C, B[],* and *M[]*; (2) *simulation clock* variable *Clock*; (3) *statistics variables SumQ[], Before[],* and *AQL[]*; (4) a *random number variable*, named *R*, for generating uniform random numbers that will be used in generating *Exp (m)* and *Uni (a, b)* random variates; (5) the *event-list variable FEL*; and (6) the *candidate activity-list variable CAL*.

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 three *properties* of *Name* (event name), *Time* (scheduled event time), and K (event parameter).

The ***ActivityList class*** contains methods for manipulating the *candidate activity list CAL*, which is defined as a member variable of the *Simulator* class. The *Activity class* which is concerned about the *candidate (or influenced) activity* has two *properties: Name* (activity name) and *K* (activity parameter).

2.3 Main Program: Run method

The main program, whose pseudo-code was given in Fig. 10.9 (Section 10.2.4) of the textbook, is implemented by the *Run method* as shown below. The main program consists of five phases: (1) *Initialization* phase, (2) *Scanning* phase, (3) *Timing* phase, (4) *Executing* phase, and (5) *Statistics collection* phase.

```
public void Run(double eosTime) {
    //1. Initialization Phase
    CAL = new ActivityList();
    FEL = new EventList();
    R = new Random();
```

```
    Event nextEvent = null;

    Clock = 0;
    Execute_Initialize_routine(Clock);

    do {
       //2. Scanning Phase
       while (!CAL.IsEmpty()) {
          Activity activity = Get_Activity();
          switch (activity.Name) {
             case "CREATE":  {
                 Execute_CREATE_activity_routine(Clock); break; }
             case "SERVE": {
                 Execute_SERVE_activity_routine(Clock, activity.K);
                 break; }
          }
       }

       //3. Timing Phase
       nextEvent = Retrieve_Event();
       Clock = nextEvent.Time;

       //4. Executing Phase
       switch (nextEvent.Name) {
          case "CREATED": { Execute_CREATED_event_routine(); break; }
          case "SERVED": { Execute_SERVED_event_routine(nextEvent.K);
                           break; }
       }
    } while (Clock < eosTime);

    //5. Statistics Collection Phase
    Execute_Statistics_routine(Clock);
}
```

In the ***Initialization*** phase of the main program, (1) member variables (*CAL*, *FEL*, and *R*) and local variables (*nextEvent*) are declared, (2) the *simulation clock* is set to zero, and (3) the initialization method *Execute_Initialize_routine ()* is invoked. As shown below, the initialization routine initializes the state variables (C = 1; B[] = 0; M[] = 1) and statistics variables (Before[] = 0; SumQ[] = 0) and stores the initially enabled activity into the CAL by invoking *Store_Activity (“CREATE”)*.

```
private void Execute_Initialize_routine(double clock)  {
    //Initialize state variables and statistics variables
    C = 1;
    B = new int[4]; M = new int[4];
    Before = new double[4]; SumQ = new double[4];

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

    //Store the initially enabled activity into CAL
    Store_Activity("CREATE");
}
```

In the ***Scanning*** phase, all the candidate activities stored in the CAL are retrieved one by one by invoking the list-handling method *Get_Activity ()* and the respective activity routine is executed. Details of the activity routines will be given shortly.

In the ***Timing*** phase, a next event is retrieved by invoking the list-handling method *Retrieve_event ()* and the simulation clock is updated; in the ***Executing*** phase, the event routine for the retrieved event is executed. Details of the event routine will also be given shortly.

Finally, in the ***Statistics collection*** phase, the AQL (average queue length) for each stage is obtained by invoking the method Execute_Statistics_routine which is defined as below:

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

2.4 Activity Routines

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

(a) `Execute_CREATE_activity_routine (clock)` and

(b) `Execute_SERVE_activity_routine (clock, k)`.

An activity routine is a subprogram that describes the changes in the state variables made at the beginning of an activity and schedules its BTO event into the FEL. An activity routine is required for each activity in the activity transition table and has the following structure: (1) Check the *At-begin condition*, (2) execute the *At-begin action* and schedule the *BTO event* of the activity if the at-begin condition is satisfied. The two activity routine methods invoked by the main program are programmed in C# as follows. The BTO event is scheduled by invoking the list-handling method *Schedule_Event ()*.

```
private void Execute_CREATE_activity_routine(double clock) {
    if (C>0){
        C--;

        double ta = Exp(10);
        Schedule_Event("CREATED", clock + ta);
    }
}
```

```
private void Execute_SERVE_activity_routine(double clock, int k) {
    if ((B[k] > 0) && (M[k] > 0)) {
        SumQ[k] += B[k] * (Clock - Before[k]); Before[k] = Clock;

        B[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("SERVED", clock + ts, k);
    }
}
```

2.5 Event Routines

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

(a) `Execute_CREATED_event_routine ()` and

(b) `Execute_SERVED_event_routine (k)`.

An event routine is a subprogram describing the changes in state variables made at the end of an activity and storing the influenced activities into CAL. One event routine is required for each activity in activity transition table and has the following structure: for each At-end arc, (1) execute the *At-end action* if the *At-end condition* is satisfied and (2) store the *influenced activities* into the CAL. The influenced activity is stored into the CAL by invoking the list-handling method *Store_Activity ()*.The two event routine methods invoked by the main program are programmed in C# as follows.

```
private void Execute_CREATED_event_routine() {
    if (true) {
        C++;
        Store_Activity("CREATE");
    }

    if (true) {
        SumQ[1] += B[1] * (Clock - Before[1]); Before[1] = Clock;

        B[1]++;
        Store_Activity("SERVE", 1);
    }
}
```

```
private void Execute_SERVED_event_routine(int k) {
    if (true) {
        M[k]++;
        Store_Activity("SERVE", k);
    }

    if (k < 3) {
        SumQ[k + 1] += B[k + 1] * (Clock - Before[k + 1]);
        Before[k + 1] = Clock;

        B[k + 1]++;
        Store_Activity("SERVE", k + 1);
    }
}
```

2.6 List Handling Methods

As explained above, the ACD dedicated simulator has two lists of priority queue *FEL* (*future event list*) and FIFO-queue *CAL* (*candidate activity list*). The *FEL* is implemented by *EventList* class that manages the BTO events and the *CAL* is implemented by *ActivityList* class that stores the candidate (or influenced) activities. In the *Simulator* class, *FEL* and *CAL* were defined as member variables as follows:

```
private EventList FEL;
private ActivityList CAL;
```

The ***list-handling methods for FEL*** defined in the Simulator class are: *`Schedule_Event`* and *`Retrieve_Event`* methods. The *Schedule_Event* method is invoked at the activity routines and the *Retrieve_Event* method is invoked at the timing phase of the main program. Please, note that the *Schedule_Event* method is overloaded with different method signatures: one takes *name* and *time* of an event and the other takes *additional argument* of "k", a parameter for the *SERVED* event. The three list-handling methods for *FEL* are programmed in C# as follows:

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

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

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

For manipulating the priority queue *FEL*, there are two methods: *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

The ***list-handling methods for CAL*** are: *Store_Activity* and *Get_Activity*. The *Store_Activity* method is invoked at the event routines and the *Get_Activity* method is invoked at the scanning phase of the main program. Please, note that the *Store_Activity* method is overloaded with different method signature: one takes the *name* of an activity and the other takes *additional argument* of "k", a parameter for the *SERVE* activity. The three list-handling methods for *CAL* are programmed in C# as follows:

```
private void Store_Activity(string name)
{
    CAL.AddActivity(name);
}

private void Store_Activity(string name, int k)
{
    CAL.AddActivity(name, k);
}
```

```
private Activity Get_Activity()
{
    return CAL.NextActivity();
}
```

For managing the FIFO queue *CAL*, there are also two methods: *AddActivity* and *NextActivity* methods. They are defined in the *ActivityList* class as follows:

- *AddActivity*(): adds an activity to the end of the list
- *NextActivity*(): retrieves an activity from the list

2.7 Random Variate Generation Methods

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). Also, you can run the dedicated simulator from the file system: you can find an executable file, “tstlsimulator.exe” under a folder of “Three-stage Tandem Line Simulator\bin\Debug”.

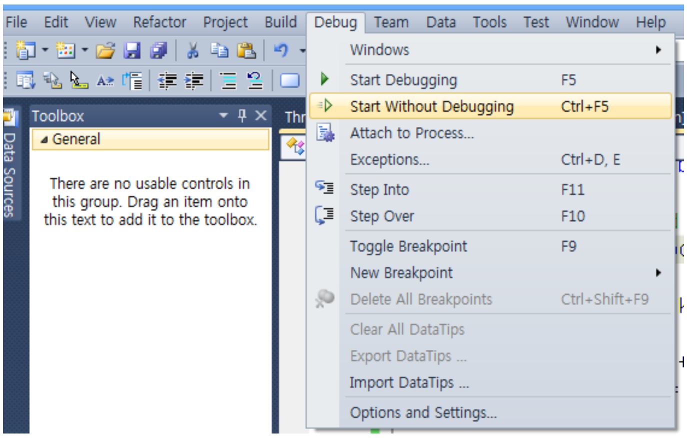
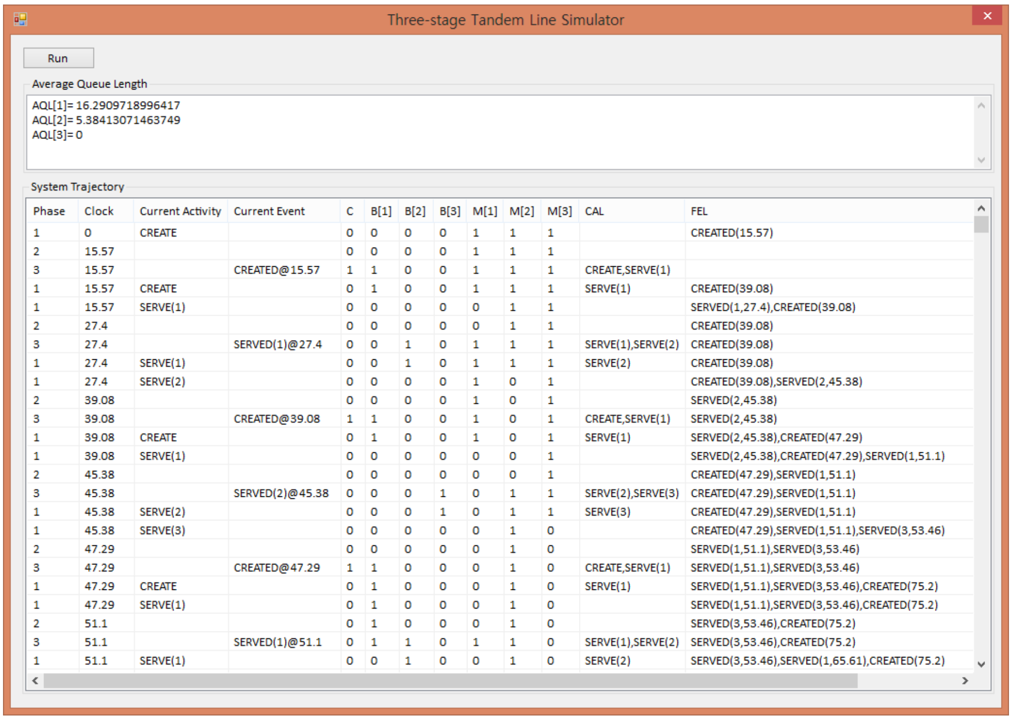


Fig. 4. Run the Dedicated Simulator from Visual Studio

If you run the dedicated simulator by clicking "Run" button, you can see the following window that displays the system trajectory (on the bottom part) together with the average queue length statistics (on the top part).


Three-stage Tandem Line Simulator

Run

Average Queue Length

AQL[1]= 16.2909718996417
AQL[2]= 5.38413071463749
AQL[3]= 0

System Trajectory

Phase	Clock	Current Activity	Current Event	C	B[1]	B[2]	B[3]	M[1]	M[2]	M[3]	CAL	FEL
1	0	CREATE		0	0	0	0	1	1	1		CREATED(15.57)
2	15.57			0	0	0	0	1	1	1		
3	15.57		CREATED@15.57	1	1	0	0	1	1	1	CREATE,SERVE(1)	
1	15.57	CREATE		0	1	0	0	1	1	1	SERVE(1)	CREATED(39.08)
1	15.57	SERVE(1)		0	0	0	0	0	1	1		SERVED(1,27.4),CREATED(39.08)
2	27.4			0	0	0	0	0	1	1		CREATED(39.08)
3	27.4		SERVED(1)@27.4	0	0	1	0	1	1	1	SERVE(1),SERVE(2)	CREATED(39.08)
1	27.4	SERVE(1)		0	0	1	0	1	1	1	SERVE(2)	CREATED(39.08)
1	27.4	SERVE(2)		0	0	0	0	1	0	1		CREATED(39.08),SERVED(2,45.38)
2	39.08			0	0	0	0	1	0	1		SERVED(2,45.38)
3	39.08		CREATED@39.08	1	1	0	0	1	0	1	CREATE,SERVE(1)	SERVED(2,45.38)
1	39.08	CREATE		0	1	0	0	1	0	1	SERVE(1)	SERVED(2,45.38),CREATED(47.29)
1	39.08	SERVE(1)		0	0	0	0	0	0	1		SERVED(2,45.38),CREATED(47.29),SERVED(1,51.1)
2	45.38			0	0	0	0	0	0	1		CREATED(47.29),SERVED(1,51.1)
3	45.38		SERVED(2)@45.38	0	0	0	1	0	1	1	SERVE(2),SERVE(3)	CREATED(47.29),SERVED(1,51.1)
1	45.38	SERVE(2)		0	0	0	1	0	1	1	SERVE(3)	CREATED(47.29),SERVED(1,51.1)
1	45.38	SERVE(3)		0	0	0	0	0	1	0		CREATED(47.29),SERVED(1,51.1),SERVED(3,53.46)
2	47.29			0	0	0	0	0	1	0		SERVED(1,51.1),SERVED(3,53.46)
3	47.29		CREATED@47.29	1	1	0	0	0	1	0	CREATE,SERVE(1)	SERVED(1,51.1),SERVED(3,53.46)
1	47.29	CREATE		0	1	0	0	0	1	0	SERVE(1)	SERVED(1,51.1),SERVED(3,53.46),CREATED(75.2)
1	47.29	SERVE(1)		0	1	0	0	0	1	0		SERVED(1,51.1),SERVED(3,53.46),CREATED(75.2)
2	51.1			0	1	0	0	0	1	0		SERVED(3,53.46),CREATED(75.2)
3	51.1		SERVED(1)@51.1	0	1	1	0	1	1	0	SERVE(1),SERVE(2)	SERVED(3,53.46),CREATED(75.2)
1	51.1	SERVE(1)		0	0	1	0	0	1	0	SERVE(2)	SERVED(3,53.46),SERVED(1,65.61),CREATED(75.2)

Fig. 5. Simulation Result at the Dedicated Simulator

In the System Trajectory, you can observe how the system state changes over time. First four columns are Phase, Clock, Current Activity, and Current Event where an Activity routine or Event routine is executed at a Clock with the phase indicating the Phase of the main program. The Phase column's value varies from 1 to 3: Phase 1 is the scanning phase (when an activity routine is executed), Phase 2 is the timing phase, and Phase 3 indicates the executing phase (where the event routine is executed). In the following seven columns represents the values of state variables, *C*, *B[]*, and *M[]*. And, the last two columns show the contents of the two lists, CAL and FEL, at the specified Clock.

4. Source Codes

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

4.1 Event.cs

```
using System;
using System.Text;

namespace MSDES.Chap10.ThreeStageTandemLine {
    /// <summary>
    /// Class for an Event Record
    /// </summary>
    public class Event {
        #region Member Variables
        private string _Name;
        private double _Time;
        private int _K;
        #endregion

        #region Properties
        /// <summary>
        /// Event Name
        /// </summary>
        public string Name { get { return _Name; } }
        /// <summary>
        /// Event Parameter (K)
        /// </summary>
        public int K { get { return _K; } }
        /// <summary>
        /// Scheduled Event Time
        /// </summary>
        public double Time { get { return _Time; } }
        #endregion

        #region Constructors
        /// <summary>
        /// Constructor for Event class
        /// </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;
            _K = int.MinValue;
        }

        /// <summary>
        /// Constructor for Event class
        /// </summary>
        /// <param name="name">The Name of an Event</param>
        /// <param name="time">The Time of an Event</param>
        /// <param name="k">Event Parameter (k)</param>
```

```
        public Event(string name, double time, int k) {
            _Name = name;
            _Time = time;
            _K = k;
        }
        #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 &&
                target.K == _K)
                rslt = true;

            return rslt;
        }

        public override string ToString() {
            if (_K == int.MinValue)
                return _Name + "@" + Math.Round(_Time, 2);
            else
                return _Name + "(" + _K + ")@" + Math.Round(_Time, 2);
        }

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

4.2 EventList.cs

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

namespace MSDES.Chap10.ThreeStageTandemLine {
    /// <summary>
    /// Container for managing events in the time-order
    /// </summary>
    public class EventList {
        #region Member Variables
        private List<Event> Events;
        #endregion

        #region Properties
        public int Count { get { return Events.Count; } }
        #endregion

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

        #region Methods
        /// <summary>
```

```
        /// Get the next event (remove the first event in the list)
        /// </summary>
        public Event NextEvent() {
            if (Events.Count == 0)  {
                throw
                  new Exception("No more event-time pair in this list");
            }

            Event nextEvent = Events[0] as Event;
            if (nextEvent == null) {
                throw new Exception("Invalid arguments, Can't find any next
event.");
            }
            Events.RemoveAt(0);
            return nextEvent;
        }

        /// <summary>
        /// Schedule an event into the future event list (FEL)
        /// </summary>
        /// <param name="name">Event Name</param>
        /// <param name="time">Event Time</param>
        public void AddEvent(string name, double time) {
            Event evt = new Event(name, time);

            if (Events.Count == 0) {
                Events.Add(evt);
                return;
            }

            for (int i = 0; i < Events.Count; i++) {
                Event item = Events[i] as Event;
                if (item != null) {
                    if (evt.Time < item.Time) {
                        Events.Insert(i, evt);
                        return;
                    }
                }
            }

            Events.Add(evt);
        }

        /// <summary>
        /// Schedule an event into the future event list (FEL)
        /// </summary>
        /// <param name="name">Event Name</param>
        /// <param name="k">Event Parameter (k)</param>
        /// <param name="time">Event Time</param>
        public void AddEvent(string name, double time, int k) {
            Event evt = new Event(name, time, k);

            if (Events.Count == 0)  {
                Events.Add(evt);
                return;
            }

            for (int i = 0; i < Events.Count; i++) {
                Event item = Events[i] as Event;
                if (item != null) {
                    if (evt.Time < item.Time)  {
```

```
                        Events.Insert(i, evt);
                        return;
                    }
                }
            }

            Events.Add(evt);
        }

        public override string ToString() {
            string str = "";

            for (int i = 0; i < Events.Count; i++) {
                Event evt = (Event)Events[i];

                if (evt.K == int.MinValue) {
                    str += evt.Name.ToString() +
                         "(" + Math.Round(evt.Time, 2).ToString() + ")";
                } else {
                    str += evt.Name.ToString() +
                  "(" + evt.K + "," + Math.Round(evt.Time, 2).ToString() + ")";
                }
                if (i < Events.Count - 1)
                    str += ",";
            }

            return str;
        }
        #endregion
    }
}
```

4.3 Activity.cs

```
using System;
using System.Text;

namespace MSDES.Chap10.ThreeStageTandemLine {
    /// <summary>
    /// Class for an Activity
    /// </summary>
    public class Activity {
        #region Member Variables
        private string _Name;
        private int _K = 0;
        #endregion

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

        /// <summary>
        /// Parameter (K)
        /// </summary>
        public int K { get { return _K; } }
        #endregion

        #region Constructors
```

```
        public Activity(string name)  {
            _Name = name;
            _K = int.MinValue;
        }

        public Activity(string name, int workstationid) {
            _Name = name;
            _K = workstationid;
        }
        #endregion

        #region Methods
        public override bool Equals(object obj)  {
            bool rslt = false;
            if (obj != null && obj is Activity)  {
                Activity target = (Activity)obj;
                if (target != null && target.Name == _Name
                    && target.K == _K)
                    rslt = true;
            }
            return rslt;
        }

        public override string ToString() {
            if (_K == int.MinValue)
                return _Name;
            else
                return _Name + "(" + _K + ")";
        }

        public override int GetHashCode() {
            return this.Name.GetHashCode();
        }

        #endregion
    }
}
```

4.4 ActivityList.cs

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

namespace MSDES.Chap10.ThreeStageTandemLine {
    /// <summary>
    /// Container for Candidate Activity List
    /// </summary>
    public class ActivityList {
        #region Member Variables
        private List<Activity> _Activities;
        #endregion

        #region Properties
        public int Count { get { return _Activities.Count; } }
        #endregion

        #region Constructors
        public ActivityList() {
            _Activities = new List<Activity>();
```

```
        }
        #endregion

        #region Methods
        /// <summary>
        /// Add a candidate activity to the end of the list
        /// </summary>
        /// <param name="name">Activity Name</param>
        public void AddActivity(string name) {
            Activity act = new Activity(name);
            _Activities.Add(act);
        }

        /// <summary>
        /// Add a candidate activity to the end of the list
        /// </summary>
        /// <param name="name">Activity Name</param>
        /// <param name="k">Activity Parameter</param>
        public void AddActivity(string name, int k) {
            Activity act = new Activity(name, k);
            _Activities.Add(act);
        }

        /// <summary>
        /// Check that the list is empty or not.
        /// </summary>
        /// <returns></returns>
        public bool IsEmpty() {
            if (_Activities.Count == 0)
                return true;
            else
                return false;
        }

        /// <summary>
        /// Retrieve next activity at the first of the list.
        /// </summary>
        /// <returns></returns>
        public Activity NextActivity() {
            if (_Activities.Count == 0)
                throw new Exception("The list is empty. No available
activities...");

            Activity act = (Activity)_Activities[0];
            _Activities.RemoveAt(0);
            return act;
        }

        public override string ToString() {
            string str = "";

            for (int i = 0; i < _Activities.Count; i++) {
                Activity activity = (Activity)_Activities[i];
                if (activity.K == int.MinValue)
                    str += activity.Name.ToString();
                else
                    str += activity.Name.ToString() + "(" + activity.K +
")";

                if (i < _Activities.Count - 1)
                    str += ",";
```

```
            }

            return str;
        }
        #endregion
    }
}
```

4.5 Simulator.cs

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

namespace MSDES.Chap10.ThreeStageTandemLine {
    public class Simulator
    {
        #region Member Variables
        /// <summary>
        /// Simulation Clock
        /// </summary>
        public double Clock;

        /// <summary>
        /// Marking for the queue C
        /// </summary>
        public int C;
        /// <summary>
        /// Markings for the parameterized queue B
        /// </summary>
        public int[] B;
        /// <summary>
        /// Markings for the parameterized queue M
        /// </summary>
        public int[] M;

        /// <summary>
        /// Candidate Actvity List
        /// </summary>
        private ActivityList CAL;
        /// <summary>
        /// Future Event List
        /// </summary>
        private EventList FEL;
        #endregion

        #region Member Variables for Collecting Statistics
        public double[] SumQ;
        public double[] Before;
        public double[] AQL;
        #endregion

        #region Member Variables for Random Variate Generation
        /// <summary>
        /// Pseudo Random Variate Generator for uniform(0,1)
        /// </summary>
        private Random R;
        #endregion

        #region Member Variables for Logging
```

```
        public string Logs;
        #endregion

        #region Constructors
        public Simulator()
        { }
        #endregion

        #region run method
        public void Run(double eosTime) {
            //1. Initialization Phase
            CAL = new ActivityList();
            FEL = new EventList();
            Logs = string.Empty;
            R = new Random();
            Event nextEvent = null;

            Clock = 0;
            Execute_Initialize_routine(Clock);

            do {
                //2. Scanning Phase
                while (!CAL.IsEmpty()) {
                    Activity activity = Get_Activity();
                    switch (activity.Name) {
                        case "CREATE":  {
                            Execute_CREATE_activity_routine(Clock);
                            break;
                        }
                        case "SERVE": {
                            Execute_SERVE_activity_routine(Clock, activity.K);
                                break;
                        }
                    }
                    Log(1, Math.Round(Clock, 2), activity.ToString(), "", C,
B[1], B[2], B[3], M[1], M[2], M[3], CAL.ToString(), FEL.ToString());
                }//end of while

                //3. Timing Phase
                //get the first event from FEL
                nextEvent = Retrieve_Event();
                //advance simulation clock
                Clock = nextEvent.Time;
                Log(2, Math.Round(Clock, 2), "", "", C, B[1], B[2], B[3],
M[1], M[2], M[3], CAL.ToString(), FEL.ToString());

                //4. Executing Phase
                switch (nextEvent.Name) {
                    case "CREATED": {
                        Execute_CREATED_event_routine();
                        break;
                    }
                    case "SERVED": {
                        Execute_SERVERD_event_routine(nextEvent.K);
                        break;
                    }
                } // end of switch-case
                Log(3, Math.Round(Clock, 2), "", nextEvent.ToString(), C,
B[1], B[2], B[3], M[1], M[2], M[3], CAL.ToString(), FEL.ToString());
            } while (Clock < eosTime);
```

```
            //5. Statistics Collection Phase
            Execute_Statistics_routine(Clock);
        }

        /// <summary>
        /// Log the current system state
        /// </summary>
        private void Log(int phase, double clock, string curActivity,
string curEvent, double c, double b1, double b2, double b3, int m1, int
m2, int m3, string cal, string fel)
        {
            Logs +=
string.Format("{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}\t{7}\t{8}\t{9}\t{10}\t{11
}\t{12}\r\n", phase, Math.Round(clock, 2), curActivity, curEvent, c, b1,
b2, b3, m1,m2, m3, cal, fel);
        }

        /// <summary>
        /// Initialize the simulation
        /// </summary>
        private void Execute_Initialize_routine(double clock) {
            //Initialize state variables and statistics variables
            C = 1;
            B = new int[4]; M = new int[4];
            Before = new double[4]; SumQ = new double[4];

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

            //Store the initially enabled activity into CAL
            Store_Activity("CREATE");
        }

        private void Execute_Statistics_routine(double clock)  {
            AQL = new double[4];
            for (int k = 1; k <= 3; k++) {
                SumQ[k] += B[k] * (clock - Before[k]);
                AQL[k] = SumQ[k] / clock;
            }
        }
        #endregion

        #region Activity List Handling Methods
        private void Store_Activity(string name) {
            CAL.AddActivity(name);
        }

        private void Store_Activity(string name, int k) {
            CAL.AddActivity(name, k);
        }

        private Activity Get_Activity() {
            return CAL.NextActivity();
        }
        #endregion

        #region Event List Handling Methods
        private void Schedule_Event(string name, double time, int k) {
            FEL.AddEvent(name, time, k);
```

```
        }

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

        private Event Retrieve_Event() {
            return FEL.NextEvent();
        }
        #endregion

        #region activity routine methods
        private void Execute_CREATE_activity_routine(double clock) {
            if (C>0){ //check the at-begin condition
                C--; //at-begin action

                //Schedule the BTO-event
                double ta = Exp(10);
                Schedule_Event("CREATED", clock + ta);
            }
        }

        private void Execute_SERVE_activity_routine(double clock, int k)
        {
            if ((B[k] > 0) && (M[k] > 0)) //check the at-begin condition
            {
                //Collect statistics
                SumQ[k] += B[k] * (Clock - Before[k]); Before[k] = Clock;

                B[k]--; M[k]--; // at-begin action

                //Schedule the BTO-event
                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("SERVED", clock + ts, k);
            }
        }
        #endregion

        #region event routine methods
        private void Execute_CREATED_event_routine() {
            if (true) {
                C++;//at-end action
                Store_Activity("CREATE"); //store influenced activity
            }

            if (true) {
                //Collect statistics
                SumQ[1] += B[1] * (Clock - Before[1]); Before[1] = Clock;

                B[1]++; //at-end action
                Store_Activity("SERVE", 1); //store influenced activity
            }
        }

        private void Execute_SERVERD_event_routine(int k) {
            if (true) {
                M[k]++; //at-end action
                Store_Activity("SERVE", k); //store influenced activity
            }
```

```
            if (k < 3) {
                //Collect statistics
                SumQ[k + 1] += B[k + 1] * (Clock - Before[k + 1]);
                Before[k + 1] = Clock;

                B[k + 1]++; //at-end action
                Store_Activity("SERVE", k + 1); //store influenced activity
            }
        }
        #endregion

        #region Random Variate Generation Methods
        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));
        }

        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
    }
}
```