A Guide to a Dedicated ACD Simulator for a Single Server System with Resource Failures

January, 2014

Byoung K. Choi and Donghun Kang

Objective

This document provides a guide to a dedicated *ACD simulator for* the *single-server system with resource failures* presented in Section 10.3.2 of the textbook *Modeling and Simulation of Discrete-Event Systems*. It gives a technical description of how the dedicated ACD simulator is implemented in C# language.

Recommendation

Prior to reading this document, the readers are recommended to read and understand Section 10.3.2 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**	**Person(s) in charge**
01/10/2014	1.0	Initial Draft	Donghun Kang <donghun.kang@kaist.ac.kr>
01/20/2014	1.1	Final Version	Byoung K. Choi <bkchoi@kaist.ac.kr>

Table of Contents

1. Introduction

Consider a *single server system* consisting of an infinite-capacity buffer and a *machine subject to random failures*. In the single server system, a new job arrives every *inter-arrival time* (t_a) and is loaded on the machine if it is idle; otherwise the job is stored in the buffer. The loaded job is processed by the machine for a period of *service time* (t_s) and then unloaded. The freed machine loads another job from the buffer if it is not empty. The machine fails every *inter-failure time* (t_f), which makes the job being processed at that time to be discarded. The failed machine is repaired by a repair man for a period of *repair time* (t_r). The *time values* associated with the single server system are distributed as follows:

- Inter-arrival time: t_a ~ Expo(5)
- Service time: t_s ~ Uniform (4, 6)
- Inter-failure time: t_f ~ Uniform (400, 450)
- Repair time: t_r ~ Uniform (5, 7)

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

1.1 ACD Model having Cancelling Arcs

Figure 1 shows an ACD model of the *single server system with resource failures* introduced in the textbook (See Fig.10.15 in Section 10.3.2 of Chapter 10), where *C* denotes the status of job creator, *Q* is the number of jobs in the buffer, *M* denotes the status of the machine (or the number of available machines in the system), *R* denotes the status of the repair man, and *E* and *F* denote the status of the *failure entity*. Table 1 is the *activity transition table* for the ACD model presented in Fig.1.

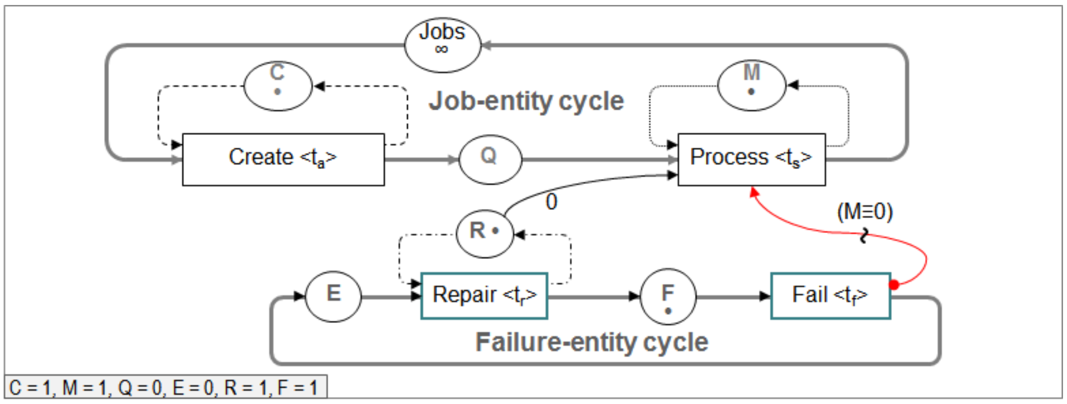


Fig.1. ACD Model having Cancelling Arcs for a Single Server System with Resource Failures

Table 1. Activity Transition Table for the Resource Failure ACD Model in Fig. 1

No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C>0)	C--;	t_a	Created	1	True	C++;	Create
						2	True	Q++;	Process
2	Process	(M>0) & (Q>0) & (R>0)	M--; Q--;	t_s	Processed	1	True	M++;	Process

3	Repair	(R>0) & (E>0)	R--; E--;	t_r	Repaired	1	True	F++;	Fail
						2	True	R++;	Repair, Process
4	Fail	(F>0)	F--;	t_f	Failed	1	True	E++;	Repair
						2	(M≡0)	M++; Cancel;	Process
Initialize		*Initial Marking* = {C=1, M=1, Q=0, E=0, R=1, F=1}; *Enabled Activities* = {Create, Fail}							

1.2 Augmented Activity Transition Table for Collecting Statistics

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

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

- SumQ = Before = 0

② SumQ and Before are updated at the (1) At-end Action entry of Create Activity, (2) At-begin Action entry of Process, and (3) At-end Action entry of Process of the ATT:

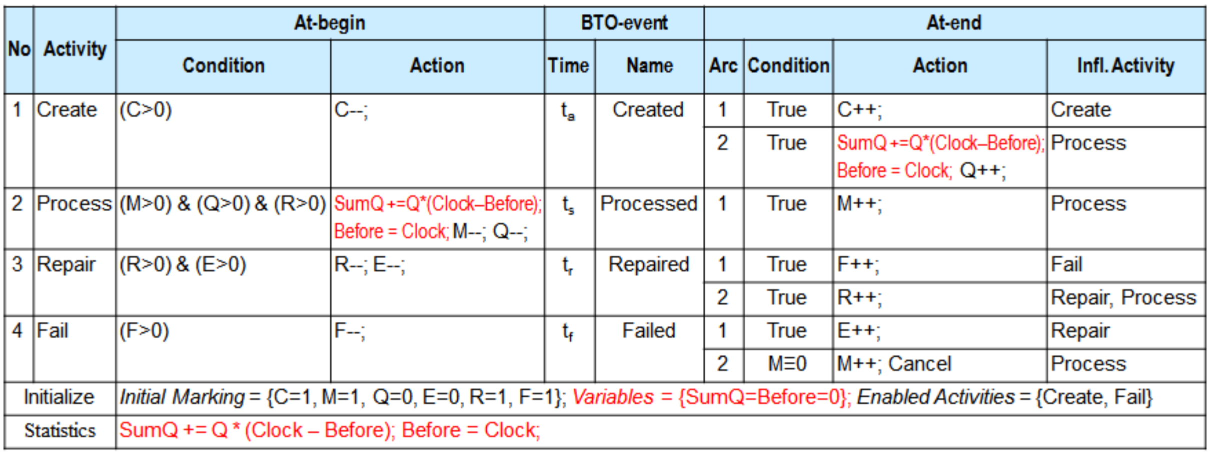
- SumQ += Q * (Clock – Before); Before = Clock;

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

- { SumQ += Q * (Clock – Before); AQL = SumQ / Clock; }

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 length



No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Infl. Activity
1	Create	(C>0)	C--;	t_a	Created	1	True	C++;	Create
						2	True	SumQ +=Q*(Clock–Before); Before = Clock; Q++;	Process
2	Process	(M>0) & (Q>0) & (R>0)	SumQ +=Q*(Clock–Before); Before = Clock; M--; Q--;	t_s	Processed	1	True	M++;	Process
3	Repair	(R>0) & (E>0)	R--; E--;	t_r	Repaired	1	True	F++;	Fail
						2	True	R++;	Repair, Process
4	Fail	(F>0)	F--;	t_f	Failed	1	True	E++;	Repair
						2	M≡0	M++; Cancel	Process
Initialize		*Initial Marking* = {C=1, M=1, Q=0, E=0, R=1, F=1}; *Variables* = {SumQ=Before=0}; *Enabled Activities* = {Create, Fail}							
Statistics		SumQ += Q * (Clock – Before); Before = Clock;							

2. Developing a Dedicated ACD Simulator

This section describes how a dedicated ACD simulator for the single server system with resource failures is developed. C# codes are based on the pseudo codes given in Section 10.2.3 of the textbook.

2.1 Development Environment

The dedicated ACD 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 ***"ResourceFailureACDSimulator.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 and open the solution file, which is named "***ResourceFailureACDSimulator.sln***".

2.2 Source Code Structure and Class Diagram

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

- `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)

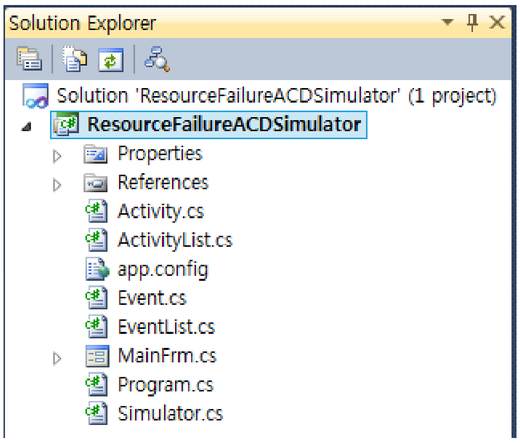


Fig.2. Source Code Structure shown in Solution Explorer of Visual Studio 2010

Figure 3 shows the class diagram consisting of the five classes: *Simulator class*, *EventList class*, *Event class*, *ActivityList class*, and *Activity class*. The *Simulator class* contains the ***main program*** (*Run*), ***activity routines*** (*Create*, *Process*, *Repair*, and *Fail*), ***event routines*** (*Created*, *Processed*, *Repaired*, and *Failed*), ***list-handling methods*** (*Store_Activity*, *Get_Activity*, *Schedule_Event*, *Retrieve_Event*, and *Cancel_Event*), 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 Express or Microsoft Visual Studio Express 2012 for Windows Desktop. The Microsoft Visual Studio Express 2012 for Windows Desktop can be downloaded freely at the following URL:
http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-for-windows-desktop

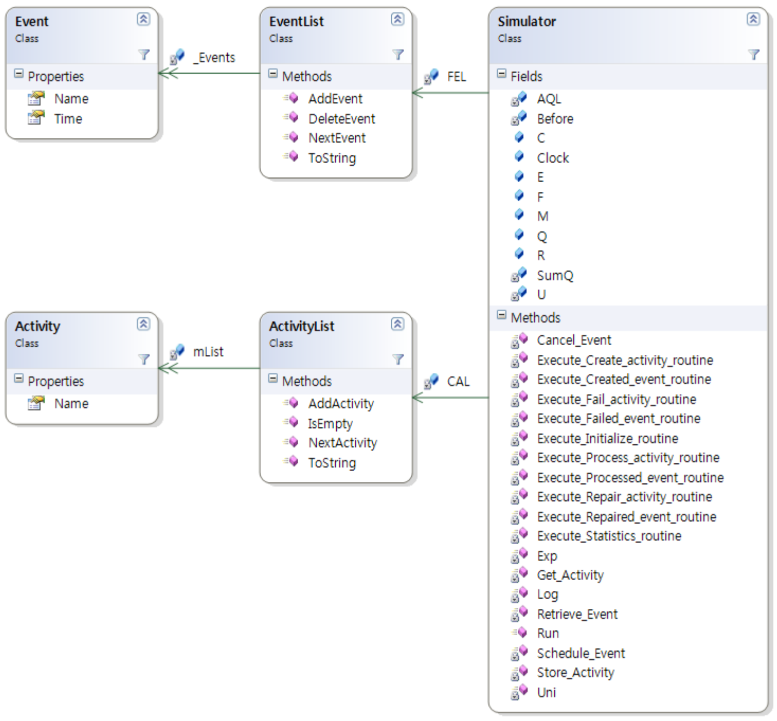


Fig.3. Class Diagram of the Dedicated Simulator

The ***member variables*** in the *Simulator* class include: 1) *state variables C*, *M*, *Q*, *E*, *R*, and *F*; 2) *simulation clock* variable *Clock*; 3) *statistics variables SumQ*, *Before*, and *AQL*; 4) a *random number variable*, named *U*, for generating uniform random numbers which will be used in generating *Exp (m)* and *Uni (a, b)* random variates; 5) the *event-list variable FEL*; and 6) the *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 two *properties* of *Name* (event name) and *Time* (scheduled event time).

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* is about the *candidate (or influenced) activity* and has a *property* of *Name* (activity name).

2.3 Main Program: *Run* method

The main program, whose pseudo-code was given in Fig. 10.6 (Section 10.2.3) 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();
    U = new Random();
    Event nextEvent = null;

    Clock = 0;
    Execute_Initialize_routine(Clock);
    do {
       //2. Scanning phase
       while (!CAL.IsEmpty()) {
          string ACTIVITY = Get_Activity();
          switch (ACTIVITY) {
              case "Create": {
                 Execute_Create_activity_routine(Clock); break; }
              case "Process": {
                 Execute_Process_activity_routine(Clock); break; }
              case "Repair": {
                 Execute_Repair_activity_routine(Clock); break; }
              case "Fail": {
                 Execute_Fail_activity_routine(Clock); break; }
          }
       }//end of while

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

       //4. Executing phase
       switch (nextEvent.Name) {
          case "Created": {
              Execute_Created_event_routine(); break; }
          case "Processed": {
              Execute_Processed_event_routine(); break; }
          case "Repaired": {
              Execute_Repaired_event_routine(); break; }
          case "Failed": {
              Execute_Failed_event_routine(); break; }
       } // end of switch-case
    } 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 *U*) 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; M= 1; Q= 0; E= 0; R= 1; F=1) and statistics variables (Before= 0; SumQ= 0) and stores the initially enabled activities into the CAL by invoking *Store_Activity ("Create")* and *Store_Activity ("Fail")*.

```
private void Execute_Initialize_routine(double Now)
{
    //Initialize state variables (markings for queues)
    C = 1; M = 1; Q = 0; E = 0; R = 1; F = 1;

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

    //store the initially enabled activity into CAL
    Store_Activity("Create");
    Store_Activity("Fail");
}
```

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) is obtained by invoking the method Execute_Statistics_routine which is defined as below:

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

2.4 Activity Routines

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

(a) `Execute_Create_activity_routine (clock)`,

(b) `Execute_Process_activity_routine (clock)`,

(c) `Execute_Repair_activity_routine (clock)`, and

(d) `Execute_Fail_activity_routine (clock)`.

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 (bound-to-occur) 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 3) schedule the *BTO event* of the activity if the at-begin condition is satisfied. The BTO event is scheduled by invoking the list-handling method *Schedule_Event ()*. The four activity routine methods (*Create*, *Process*, *Repair, and Fail*) invoked by the main program are programmed in C# as follows.

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

        double ta = Exp(5);
        Schedule_Event("Created", clock + ta);
    }
}
```

```
private void Execute_Process_activity_routine(double clock)
{
    if ((M > 0) && (Q > 0) && (R > 0)) {
        SumQ += Q * (Clock - Before); Before = Clock;
        M--; Q--;

        double ts = Uni(4, 6);
        Schedule_Event("Processed", clock + ts);
    }
}
```

```
private void Execute_Repair_activity_routine(double clock)
{
    if ((R > 0) && (E > 0)) {
        R--; E--;

        double tr = Uni(5, 7);
        Schedule_Event("Repaired", clock + tr);
    }
}
```

```
private void Execute_Fail_activity_routine(double clock)
{
    if (F > 0) {
        F--;

        double tf = Uni(400, 450);
        Schedule_Event("Failed", clock + tf);
    }
}
```

2.5 Event Routines

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

(a) `Execute_Created_event_routine (),`

(b) `Execute_Processed_event_routine (),`

(c) `Execute_Repaired_event_routine (), and`

(d) `Execute_Failed_event_routine ().`

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 ()*. If any activity needs to be canceled, its BTO event can be canceled by invoking the list-handling method *Cancel_Event ()*. The four event routine methods (*Created*, *Processed*, *Repaired*, *and Failed*) 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 += Q * (Clock - Before); Before = Clock;
        Q++;
        Store_Activity("Process");
    }
}
```

```
private void Execute_Processed_event_routine()
{
    if (true) {
        M++;
        Store_Activity("Process");
    }
}
```

```
private void Execute_Repaired_event_routine()
{
    if (true) {
        R++;
        Store_Activity("Repair");
        Store_Activity("Process");
    }

    if (true) {
        F++;
        Store_Activity("Fail");
    }
}
```

```
private void Execute_Failed_event_routine()
{
    if (true) {
        E++;
        Store_Activity("Repair");
    }

    if (M == 0) {
        M++;
        Cancel_Event("Processed");
    }
}
```

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*** in the Simulator class are: `Schedule_Event (name, time)`, `Retrieve_Event ()`, and `Cancel_Event ()`. The *Schedule_Event* method is invoked at the activity routines, the *Retrieve_Event* method is invoked at the timing phase of the main program, and the *Cancel_Event* method is invoked at the event routines if their at-end action has a “Cancel”. 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 Event Retrieve_Event()
{
    Event nextEvent = null;
    nextEvent = FEL.NextEvent();
    return nextEvent;
}
```

```
private void Cancel_Event(string name)
{
    FEL.DeleteEvent(name);
}
```

For manipulating the priority queue *FEL*, there are two methods: *AddEvent*, *DeleteEvent*, 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)
- *DeleteEvent*(): delete an event from the list whose name is same as the specified name with the least time value
- *NextEvent*(): retrieves a next event next from the list

The ***list-handling methods for CAL*** are: `Store_Activity (name)` 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. The four list-handling methods for *CAL* are programmed in C# as follows:

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

```
private string Get_Activity()
{
    Activity act = CAL.NextActivity();
    return act.Name;
}
```

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 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 = U.NextDouble();
    return (a + (b - a) * u);
}
```

U.NextDouble () method returns a random number between 0.0 and 1.0. As mentioned in

Section 2.2, “U” is a member variable of the *Simulator* class, which is a pseudo-random number generator (*System.Random* class) provided by C# language.

```
private Random U;
```

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 = U.NextDouble();
    return (-a * Math.Log(u));
}
```

3. Simulation Execution

If you want to run the dedicated simulator from Visual Studio 2010, click the menu item *Debug* > *Start Without Debugging* (or click the short key, Ctrl + F5) as shown in Figure 4. You can also run the dedicated simulator from the file system: you can find an executable file, “simulator.exe” under a folder of “ResourceFailureACDSimulator\bin\Debug”.

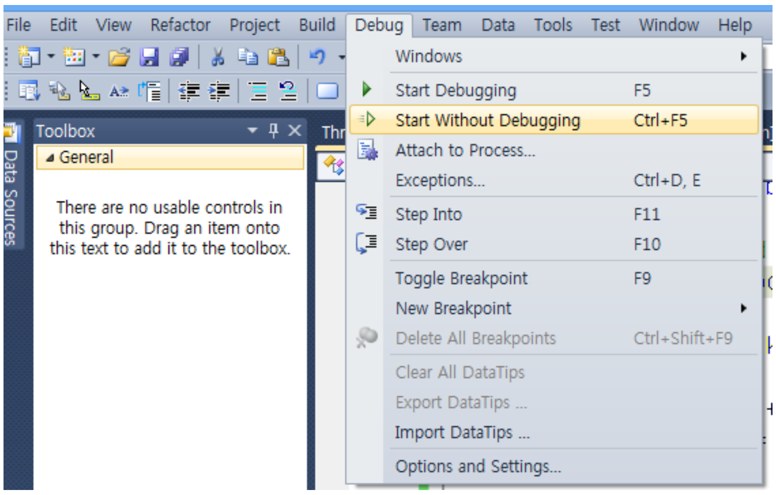
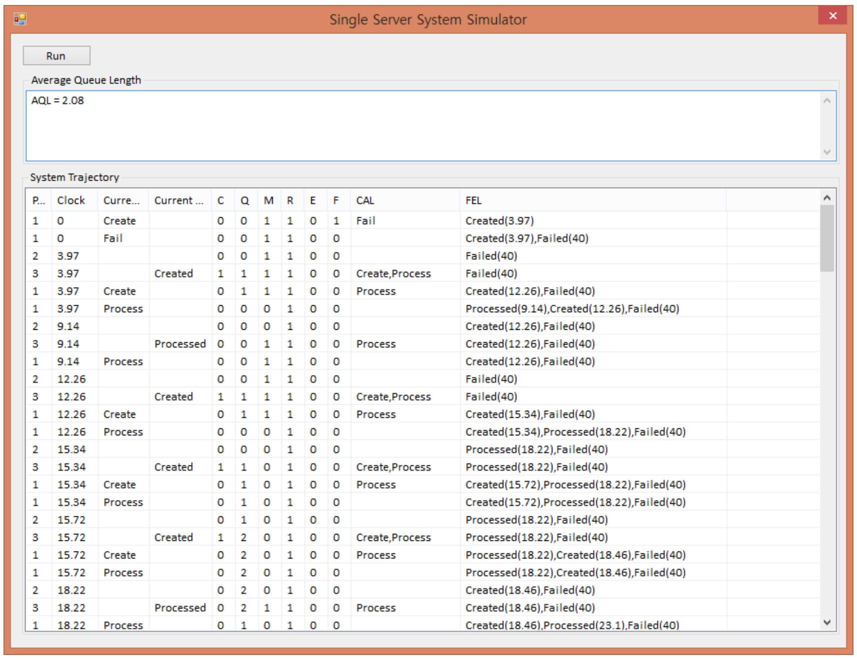


Fig. 4. Run the Dedicated Simulator from Visual Studio

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).



AQL = 2.08

P...	Clock	Curre...	Current ...	C	Q	M	R	E	F	CAL	FEL
1	0	Create		0	0	1	1	0	1	Fail	Created(3.97)
1	0	Fail		0	0	1	1	0	0		Created(3.97),Failed(40)
2	3.97			0	0	1	1	0	0		Failed(40)
3	3.97		Created	1	1	1	1	0	0	Create,Process	Failed(40)
1	3.97	Create		0	1	1	1	0	0	Process	Created(12.26),Failed(40)
1	3.97	Process		0	0	0	1	0	0		Processed(9.14),Created(12.26),Failed(40)
2	9.14			0	0	0	1	0	0		Created(12.26),Failed(40)
3	9.14		Processed	0	0	1	1	0	0	Process	Created(12.26),Failed(40)
1	9.14	Process		0	0	1	1	0	0		Created(12.26),Failed(40)
2	12.26			0	0	1	1	0	0		Failed(40)
3	12.26		Created	1	1	1	1	0	0	Create,Process	Failed(40)
1	12.26	Create		0	1	1	1	0	0	Process	Created(15.34),Failed(40)
1	12.26	Process		0	0	0	1	0	0		Created(15.34),Processed(18.22),Failed(40)
2	15.34			0	0	0	1	0	0		Processed(18.22),Failed(40)
3	15.34		Created	1	1	0	1	0	0	Create,Process	Processed(18.22),Failed(40)
1	15.34	Create		0	1	0	1	0	0	Process	Created(15.72),Processed(18.22),Failed(40)
1	15.34	Process		0	1	0	1	0	0		Created(15.72),Processed(18.22),Failed(40)
2	15.72			0	1	0	1	0	0		Processed(18.22),Failed(40)
3	15.72		Created	1	2	0	1	0	0	Create,Process	Processed(18.22),Failed(40)
1	15.72	Create		0	2	0	1	0	0	Process	Processed(18.22),Created(18.46),Failed(40)
1	15.72	Process		0	2	0	1	0	0		Processed(18.22),Created(18.46),Failed(40)
2	18.22			0	2	0	1	0	0		Created(18.46),Failed(40)
3	18.22		Processed	0	2	1	1	0	0	Process	Created(18.46),Failed(40)
1	18.22	Process		0	1	0	1	0	0		Created(18.46),Processed(23.1),Failed(40)

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

In the System Trajectory, you can observe how the system state changes over time. The 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 value of the 'Phase' column 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 six columns represents the values of state variables, *C*, *Q*, *M*, *R*, *E*, and *F*. 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. ResourceFailure {
    /// <summary>
    /// Class for an Event Record
    /// </summary>
    public class Event {
        #region Member Variables
        private string _Name;
        private double _Time;
        #endregion
        #region Properties
        /// <summary>
        /// Event Name
        /// </summary>
        public string Name { get { return _Name; } }

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

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

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

            return rslt;
        }

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

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

4.2 EventList.cs

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

namespace MSDES.Chap10.ResourceFailure
{
    /// <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() {
            Event next_event = null;
            if (_Events.Count > 0) {
                next_event = _Events[0];
                _Events.RemoveAt(0);
            }
            return next_event;
        }

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

            if (_Events.Count == 0) {
                _Events.Add(nextEvent);
            } else {
                bool isAdded = false;
                for (int i = 0; i < _Events.Count; i++) {
                    Event e = _Events[i];
                    if (nextEvent.Time <= e.Time) {
                        _Events.Insert(i, nextEvent);
                        isAdded = true;
                        break;
                    }
                }
                if (!isAdded) _Events.Add(nextEvent);
            }
        }

        /// <summary>
        /// Delete an event from the list whose name is same as the
        /// specified name with the least time value.
        /// </summary>
        /// <param name="eventName">Event Name</param>
        public void DeleteEvent(string eventName) {
            for (int i = 0; i < _Events.Count; i++) {
                if (_Events[i].Name == eventName) {
                    _Events.RemoveAt(i);
                    break;
                }
            }
        }
        public override string ToString() {
            string str = "";
            for (int i = 0; i < _Events.Count; i++) {
                Event evt = (Event)_Events[i];
                str += evt.Name.ToString() +
                          "(" + 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.ResourceFailure {
    /// <summary>
    /// Class for an Activity
    /// </summary>
    public class Activity {
        #region Member Variables
        private string _Name;
        #endregion

        #region Properties
        public string Name { get { return _Name; } }
        #endregion

        #region Constructors
        public Activity(string name) {
            _Name = name;
        }
        #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)
                    rslt = true;
            }
            return rslt;
        }

        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.ResourceFailure
{
    /// <summary>
    /// Container for Candidate Activity List
    /// </summary>
    public class ActivityList {
        #region Member Variables
        private List<Activity> mList;
        #endregion

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

        #region Constructors
        public ActivityList() {
            mList = new List<Activity>(); }
        #endregion

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

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

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

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

        public override string ToString() {
            string str = "";
            for (int i = 0; i < mList.Count; i++) {
                Activity activity = (Activity)mList[i];
                str += activity.Name.ToString();
                if (i < mList.Count - 1)
                    str += ",";
            }
            return str;
        }
        #endregion
    }
}
```

4.5 Simulator.cs

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

namespace MSDES.Chap10.ResourceFailure {
    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 queue Q
        /// </summary>
        public int Q;
        /// <summary>
        /// Markings for the queue M
        /// </summary>
        public int M;
        /// <summary>
        /// Markings for the queue E
        /// </summary>
        public int E;
        /// <summary>
        /// Markings for the queue R
        /// </summary>
        public int R;
        /// <summary>
        /// Markings for the queue F
        /// </summary>
        public int F;

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

        #region Member Variables for Collecting Statistics
        private double SumQ;
        private double Before;
        private double AQL;
        public double AverageQueueLength {
            get { return this.AQL; }
        }
        #endregion

        #region Member Variables for Random Variate Generation
        /// <summary>
        /// Pseudo Random Variate Generator for uniform distribution
        /// </summary>
        private Random U;
        #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;
            U = new Random();

            Clock = 0;
            Execute_Initialize_routine(Clock);

            //Simulation
            Event nextEvent = null;
            do {
                //2. Scanning phase
                while (!CAL.IsEmpty()) {
                    string ACTIVITY = Get_Activity();
                    switch (ACTIVITY) {
                        case "Create": {
                            Execute_Create_activity_routine(Clock); break; }
                        case "Process": {
                            Execute_Process_activity_routine(Clock); break; }
                        case "Repair": {
                            Execute_Repair_activity_routine(Clock); break; }
                        case "Fail": {
                            Execute_Fail_activity_routine(Clock); break; }
                    }

                    Log(1, Math.Round(Clock, 2), ACTIVITY, "", C, Q, M, R, E,
F, 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, Q, M, R, E, F,
CAL.ToString(), FEL.ToString());

                //4. Executing phase
                switch (nextEvent.Name) {
                  case "Created": {
                     Execute_Created_event_routine(); break; }
                  case "Processed": {
                     Execute_Processed_event_routine(); break; }
                  case "Repaired": {
                     Execute_Repaired_event_routine(); break; }
                  case "Failed": {
                     Execute_Failed_event_routine(); break;}
               } // end of switch-case
               Log(3, Math.Round(Clock, 2), "", nextEvent.Name, C, Q, M, R,
E, F, 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 q, int m, int r, int e, int f, 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
}\r\n", phase, Math.Round(clock, 2), curActivity, curEvent, c, q, m, r, e,
f, cal, fel);
        }

        #endregion

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

        private string Get_Activity() {
            Activity act = CAL.NextActivity();
            return act.Name;
        }

        #endregion

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

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

        private void Cancel_Event(string name) {
            FEL.DeleteEvent(name);
        }
        #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

                double ta = Exp(5);
                Schedule_Event("Created", clock + ta); //Schedule the BTO-
event
            }
        }

        private void Execute_Process_activity_routine(double clock) {
            if ((M > 0) && (Q > 0) && (R > 0)) { //check the at-begin
condition
                SumQ += Q * (Clock - Before); Before = Clock; //Collect
statistics

                M--; Q--;// at-begin action
                double ts = Uni(4, 6);
                Schedule_Event("Processed", clock + ts); //Schedule the BTO-
event
            }
        }

        private void Execute_Repair_activity_routine(double clock) {
            if ((R > 0) && (E > 0)) { //check the at-begin condition
                R--; E--; // at-begin action
                double tr = Uni(5, 7);
                Schedule_Event("Repaired", clock + tr); //Schedule the BTO-
event
            }
        }

        private void Execute_Fail_activity_routine(double clock)  {
            if (F > 0) { //check the at-begin condition
                F--; // at-begin action
                double tf = Uni(400, 450);
                Schedule_Event("Failed", clock + tf); //Schedule the BTO-
event
            }
        }
        #endregion

        #region event routine methods
        private void Execute_Initialize_routine(double clock) {
            //Initialize state variables (markings for queues)
            C = 1; M = 1; Q = 0; E = 0; R = 1; F = 1;

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

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

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

        private void Execute_Created_event_routine() {
            if (true) {
                C++;//at-end action
                Store_Activity("Create"); //store influenced activity
            }

            if (true) {
                SumQ += Q * (Clock - Before); Before = Clock;//Collect
statistics

                Q++; //at-end action
                Store_Activity("Process"); //store influenced activity
            }
        }

        private void Execute_Processed_event_routine() {
            if (true) {
                M++; //at-end action
                Store_Activity("Process"); //store influenced activity
            }
        }

        private void Execute_Repaired_event_routine() {
            if (true) {
                R++; //at-end action
                Store_Activity("Repair"); //store influenced activity
                Store_Activity("Process"); //store influenced activity
            }

            if (true) {
                F++; //at-end action
                Store_Activity("Fail"); //store influenced activity
            }
        }

        private void Execute_Failed_event_routine() {
            if (true) {
                E++; //at-end action
                Store_Activity("Repair"); //store influenced activity
            }

            if (M == 0) {
                M++; //at-end action
                Cancel_Event("Processed"); //at-end action
            }
        }
        #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 = U.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 = U.NextDouble();
            return (a + (b - a) * u);
        }
        #endregion
    }
}
```