Proceedings of the 2014 Winter Simulation Conference
A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, eds.

HOW TO DEVELOP YOUR OWN SIMULATORS FOR DISCRETE-EVENT SYSTEMS

Byoung K. Choi
Donghun Kang

Department of Industrial and Systems Engineering
Korea Advanced Institute of Science and Technology (KAIST)
Daehak-ro (373-1 Guseong-dong), Yuseong-gu
Daejeon, 305-701, REPUBLIC OF KOREA

ABSTRACT

This tutorial explains how to develop dedicated simulators for executing event graph models and activity cycle diagram (ACD) models. An event-graph simulator template and an ACD simulator template are presented in pseudo code form, together with example C# implementations for a simple discrete-event system. A list of the simulation programs in C# codes is provided in a website. A brief description of a general-purpose simulator for executing ACD models is also presented.

1 INTRODUCTION

A discrete-event system (DES) consisting of resources (e.g. machines and buffers) and entities (e.g. jobs) is often referred to as a *regular DES*. A DES without physical resources may be termed a *resource-less DES*. The entities in a resource-less DES are referred to as *agents*, and the basic concept of *agent-based* modeling is that a system is modeled through placing agents in the system and letting the system evolve from the interaction of those agents. Regular DESs include most service and manufacturing systems that are designed and built, as well as traffic systems and military systems. A flock of seasonal birds in the sky is an example of resource-less DES. Among the DES modeling formalisms (e.g. Petri nets, DEVS, timed automata, event graphs, and activity cycle diagrams), the event graph and activity cycle diagram (ACD) are commonly used in simulation modeling (i.e. modeling for simulation) of regular DESs, whereas Petri nets and timed automata are primarily used in modeling DESs for analysis purposes. This tutorial explains how to develop dedicated simulators for executing event graph models and ACD models. A brief description of a general purpose simulator ACE® for executing ACD models is also presented. To the best of the authors' knowledge, there are no other tools that can be used to directly execute ACD models.

1.1 Event Graph

The event-based modeling concept was realized in SIMSCRIPT II in the 1960s (Kiviat, Villanueva, and Markowitz 1968), but the event graph formalism was established later by Schruben (1983). An ***event graph*** is a *directed graph* ($\boldsymbol{G}_{EG}$ = <*V, E, S, F, C, D, A*>) defined by a set of *vertices* ($V = \{v\}$) that represents the *events,* a set of directed *edges* ($E = \{e_{od} = (v_o, v_d)\}$) that represents the temporal and logical *relationships* between pairs of events, and a set of *state variables* ($S = \{s\}$) that represents the system state. Associated with each vertex (v) is a *state update function* ($f_v \in F$) that describes the state change caused by the event. Associated with $E = \{e\}$ are a set of *conditions* ($C = \{c_e\}$), a set of *time delays* ($D = \{d_e\}$), and a set of *action types* ($A = \{a_e \in \{\text{scheduling, canceling}\}\}$). A state-of-the-art review of event graph is given in Savage, Schruben, and Yücesan (2005).

Figure 1 presents an event graph model for a single server system initially consisting of an idle *machine* (M = 1) and an empty *buffer* (Q = 0): (1) an Arrive event, which increases the job count (Q) in the

buffer, always schedules another Arrive event to occur after t_a and schedules a Load event to occur immediately if the machine is idle (M ≡ 1); (2) a Load event, which sets the machine to busy (M--) and decreases the job count by one (Q--), schedules an Unload event to occur after t_s; and (3) the Unload event resets the machine to idle (M++) and schedules a Load event if the buffer is not empty (Q > 0). Thus, the single server event graph model may be specified as follows: $V = \{v_1 = \text{Arrive}, v_2 = \text{Load}, v_3 = \text{Unload}\}$, $E = \{e_{11} = (v_1, v_1), e_{12} = (v_1, v_2), e_{23} = (v_2, v_3), e_{32} = (v_3, v_2)\}$, $S = \{\text{Q, M}\}$, etc.

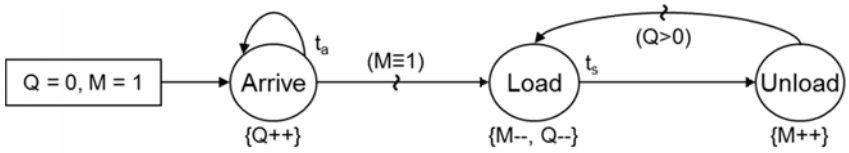


Figure 1: Event graph model of a single server system.

1.2 Activity Cycle Diagram

The ACD was invented by Tocher in the late 1950s in order to solve a congestion control problem at a steel mill (Hollocks 2008). Tocher used a *flow diagram of activities* to model the dynamic behavior of the steel plant. The *activity flow diagram* later evolved into the *classical ACD* where an *activity node* is denoted using a rectangle and a *queue node* is denoted using a circle (Carrie 1988). A ***classical ACD*** model is a *bipartite directed graph* ($\boldsymbol{G}_{ACD} = \langle A, B, E, T, \boldsymbol{\mu}, \boldsymbol{\mu}_0 \rangle$) defined by a finite set of activity nodes ($A = \{a_i\}$), a finite set of queue nodes ($B = \{b_j\}$), a finite set of edges ($E = \{e_k\}$ with $e_k \in (A \times B) \cup (B \times A)$), a set of time delay functions ($T = \{\tau_a\}$) associated with $a \in A$, a marking vector ($\boldsymbol{\mu} = \{\mu_b\}$) associated with $b \in B$, and an initial marking $\boldsymbol{\mu}_0$. A state-of-the-art review of ACD is given in Choi et al. (2013).

Figure 2 presents a classical ACD model of the same single server system as depicted in Figure 1 where the activity nodes are Create and Process; the queue nodes are C, Jobs, Q, and M; t_a is the inter-arrival time; and t_s is the service time. In the figure, the main solid-line loop denotes the *entity-activity cycle* and the dashed line loops denote the *resource-activity cycles* (Creator cycle and Machine cycle). Thus, the single server ACD model may be specified as follows: $A = \{a_1 = \text{Create}, a_2 = \text{Process}\}$, $B = \{b_1 = \text{Jobs}, b_2 = \text{C}, b_3 = \text{Q}, b_4 = \text{M}\}$ … $\boldsymbol{\mu}_0 = \{\infty, 1, 0, 1\}$.

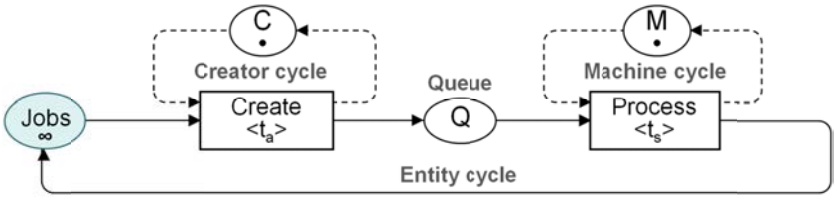


Figure 2: ACD model of a single server system.

2 HOW TO DEVELOP YOUR OWN DEDICATED EVENT GRAPH SIMULATORS

This section will assist you in developing your own simulation programs for executing a given event graph model using the *next event scheduling algorithm*. It is assumed that you have basic skills in computer programming. The process of developing a dedicated simulator for a given event graph model will be described in a *bottom-up manner*, starting from the *primitive functions for handling events*, followed by functions for generating random variates.

2.1 Functions for Handling Events

In order to simulate the system dynamics of a discrete event system, we need a mechanism for processing future events. A *future event* is an event that has been scheduled to occur in the future; a future event that has the smallest (i.e. earliest) event-time is called the ***next event***. The simulation maintains a *future event list* (FEL), which is an *ordered list* of pairs ({<E_k, T_k>}), where T_k is the scheduled execution time of the *future event* (E_k). The FEL is also a *priority queue*, ordered in increasing values of T_k.

Figure 3 presents a schematic description of the three ***event-handling functions***: Schedule-event (), Retrieve-event (), and Cancel-event (). Initially, there are three *future events* {<E1, 12.1>, <E2, 18.6>, <E3, 34.0>} stored in the FEL. The mechanisms of the event-handling functions are:

1. If the Schedule-event (E4, 22.7) function is invoked, the *incoming event* <E4, 22.7> is inserted immediately after the incumbent event <E2, 18.6> in the FEL, which is a priority queue in an increasing order of *event times*. Now, FEL has four *future events*: <E1, 12.1>, <E2, 18.6>, <E4, 22.7>, and <E3, 34.0>. If an incoming event has the same event time as an incumbent event, the former is stored right after the latter so that a FIFO rule is applied (Other tie-breaking rules are possible).
2. If the Retrieve-event (E, T) function is invoked, the *next event* <E = E1, T = 12.1> is retrieved (and deleted from the FEL).
3. If the Cancel-event (E4) function is invoked, the event node <E4, 22.7> is deleted from the FEL.

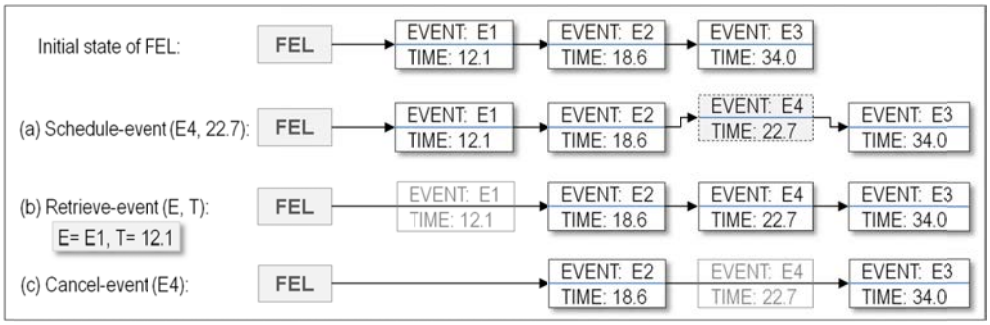


Figure 3: Schematic descriptions of the event handling functions.

2.2 Functions for Generating Random Variates

Most programming languages support a built-in function for generating a standard *uniform random number* ($u \sim U[0,1]$). In Java, for example, the function u = Math.random() does the job. Let $x \sim U[a, b]$, then x is obtained from u as follows: $x = a + (b-a)*u$. An *exponential random variate* (X) is generated from a *uniform random number* (U) as follows. Because the distribution function F(X) can be regarded as a uniform random number (U), we have $U = F(X) = 1 - e^{-x/\theta}$, where θ is the mean. Upon solving this equation for X, we can obtain $X = -\theta*\ln(1-U)$, which is equivalent to $X = -\theta*\ln(U)$ because (1 – U) is also a uniform random number. This method of generating a random variable is referred to as the *inverse transformation* method. In Java, the natural log (ln(U)) is implemented as Math.log (u). The Java and C# codes for generating exponential random variates and uniform random variates are listed in Figure 4. A detailed description on the subject is provided in Law (2007).

2.3 Event Routines

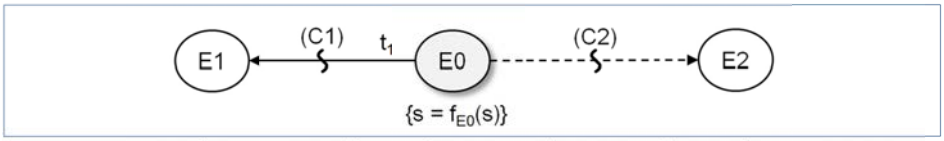
Figure 5 presents a portion of an event graph for an *event vertex* that has a scheduling edge and a canceling edge. The event graph indicates that "*whenever E0 occurs, the state variable s changes to* $f_{E0}(s)$.

```
(a) public double Exp(double a) { if (a<=0) return -1;
      double u=Math.random(); return (-a*Math.log(u)); }
    public double Uni(double a, double b) {
      if (a>=b) return -1;
      double u=Math.random(); return (a+(b- a)*u); }
```

```
(b) Random U = new Random();
    public double Exp(double a) { if (a<=0) return -1;
      double u=U.NextDouble(); return (-a*Math.Log(u));}
    public double Uni(double a, double b) { if (a >= b) return -1;
      double u=U.NextDouble(); return (a + (b-a)*u);}
```

Figure 4: Random variate generation functions in (a) Java and (b) C# .

Then, if edge condition C1 is true, E1 is scheduled to occur after t_1; if edge condition C2 is true, E2 is canceled immediately". Figure 5 also illustrates an event transition table for E0. An ***event transition table*** is a tabular form of formally specifying an event graph model. Specified for each originating event are the state change, edge conditions, action types (schedule/cancel), time delays, and destination events.



Originating Event	State Change	Edge	Condition	Action	Delay	Destination Event
E0	s = f_{E0}(s)	1	C1	schedule	t_1	E1
		2	C2	cancel	0	E2

Figure 5: Originating event vertex E0 with a scheduling edge and canceling edge.

An ***event routine*** for an *originating event* is a subprogram that describes the changes in the state variables and how the *future events* are scheduled and/or canceled. Conditional events are handled inside each event routine. One event routine is required for each event in the event graph model. The event routine for E0 in Figure 5 is expressed as follows:

Execute-E0-event-routine (Now) { s = f_{E0}(s); If (C1) Schedule-event (E1, Now+ t_1); If (C2) Cancel-event (E2) }.

2.4 Next Event Scheduling Algorithm for Simulation Execution

The overall procedure of the simulation execution, which is called the *next event scheduling algorithm*, is as follows: (0) Reset the *simulation clock* (CLK); (1) initialize *state variables* and schedule *initial events*; (2) retrieve the next event <E, T> from the FEL and set CLK to next event time (T); (3) execute the event routine for the next event (E); (4) if a termination condition is satisfied, compute the output statistics, otherwise return to step (2). The above *next event scheduling algorithm* may be drawn as a flow chart as seen in Figure 6.

In Figure 6, the simulation is terminated if an *end of simulation* (EOS) condition is met. Figure 7 describes the *main program* template of the event graph simulator that implements the next event scheduling algorithm. The "*Initialize*" box and "*Output statistics*" box in Figure 6 are implemented as an *initialize routine* and a *statistics routine*, respectively, in Figure 7. The simulation is performed inside the *While* loop. Listed in the *Event-routines list* are the *event routines* for E1~En. Here, the EOS condition is specified using the simulation clock (CLK) and EOS time (t_e).

2.5 Single Server System Event Graph Simulator

The *process of programming a dedicated simulator* for a given event graph model is as follows: (1) the (pure) event graph model is converted to an *augmented event graph model* through adding the statistics variables and a statistics routine; (2) an event transition table is constructed from the augmented

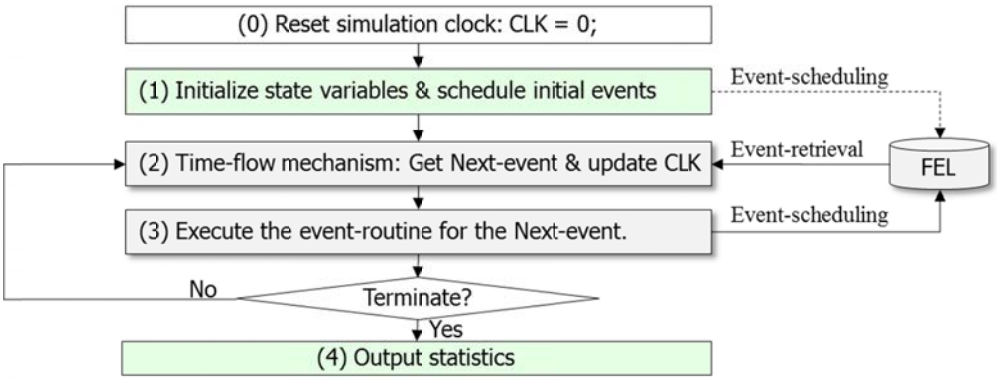


Figure 6: Next event scheduling algorithm.

```
Begin
    CLK = 0;
    Execute-initialize-routine (CLK);
    While (CLK < te) do {                                  // (1) Initialize
        Retrieve-event (EVENT, TIME); CLK = TIME;          // (2) Time-flow mechanism
        Case EVENT of {                                    // (3) Execute event-routine
            E1:  Execute-E1-event-routine (CLK);
            E2:  Execute-E2-event-routine (CLK);
            ....
            En:  Execute-En-event-routine (CLK);           - Event-routines list
        } // end-of-case
    }; // end-of-while
    Execute-statistics-routine (CLK);                      // (4) Output statistics
End
```

Figure 7: Main program template implementing the next event scheduling algorithm.

event graph model; (3) the initialize routine, event routines, and statistics routine are developed; and (4) the main program is obtained from the main program template given in Figure 7. The process of constructing your own event graph simulator will be described using the *single server system* model given in Figure 1. It is assumed that the EOS time (t_e) is 500 and the distributions of the *inter-arrival times* (t_a) and *service times* (t_s) are *Exp (5)* and *Uni (4, 6)*, respectively.

Figure 8 presents an *augmented event graph* for collecting the *average queue length* (AQL) statistics. Let $\{C_k\}$ denote the *queue length change times*, then the k^{th} *queue length change interval* becomes $\Delta k = C_{k+1} - C_k$. Let Q_k be the queue size during Δk, then the AQL is expressed as $AQL = \Sigma(Q_k \times \Delta k) / \Sigma(\Delta k) \equiv SumQ / CLK$. Here, two *statistics variables* are introduced: SumQ for accumulating the queue length values over time and Before for the previous event time. An *event transition table* for this event graph model in Figure 8 is given in Table 1.

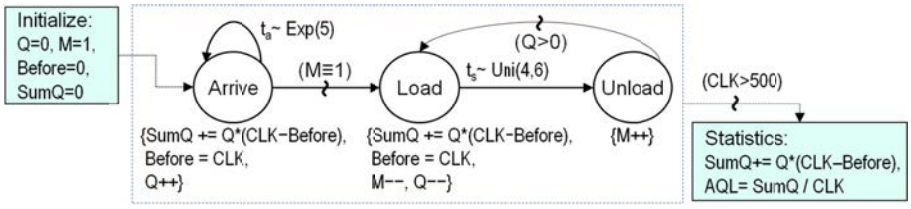


Figure 8: Augmented event graph model for collecting AQL statistics.

The initialize routine, event routines, and statistics routine of the augmented event graph model for collecting the AQL statistics are listed in Figure 9. Then, using the main program template in Figure 7, the main program in pseudo-codes of the *single server system event graph simulator* is obtained as described in Figure 10. A C# implementation of the pseudo-codes is presented in Section 2.6.

Table 1: Event transition table for the event graph model of Figure 8.

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

```
Execute-Initialize-routine (Now) { Q = 0; M = 1; Before = 0; SumQ = 0; Schedule-event (Arrive, Now); }
Execute-Arrive-event-routine (Now) { SumQ += Q*(Now – Before); Before = Now; Q++;
  Schedule-event (Arrive, Now+ Exp (5)); if (M≡1) Schedule-event (Load, Now); }
Execute-Load-event-routine (Now) { SumQ += Q*(Now – Before); Before = Now; M−−; Q−−;
  Schedule-event (Unload, Now+ Uni (4, 6)); }
Execute-Unload-event-routine (Now) { M++;
  if (Q>0) Schedule-event (Load, Now); }
Execute-statistics-routine (Now) { SumQ += Q*(Now – Before); AQL = SumQ / Now; }
```

Figure 9: Initialize routine, event routines, and statistics routine for the single serve system.

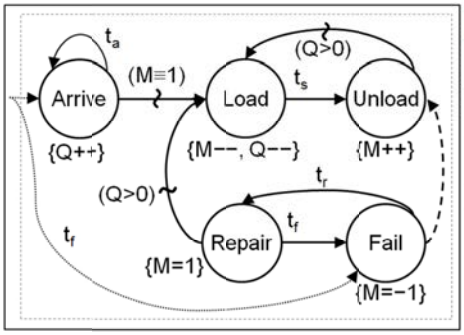
```
Begin
    CLK = 0;
    Execute-initialize-routine (CLK);
    While (CLK < 500) do {    // te = 500
        Retrieve-event (EVENT, TIME); CLK = TIME;
        Case EVENT of {
                Arrive:      Execute-Arrive-event-routine (CLK);
                Load:        Execute-Load-event-routine (CLK);
                Unload:      Execute-Unload-event-routine (CLK);
        } // end-of-case
    }; // end-of-while
    Execute-statistics-routine (CLK);
End
```

Figure 10: Main program of the single server system event graph simulator.

As another example, Figure 11 presents an *event graph model* and an *augmented event transition table* for a *single server system with resource failures* (t_f = inter-failure time; t_r = repair time). Compared with the single server system event graph of Figure 1, it has two additional event vertices: *Fail* and *Repair*. In addition, a Fail event must be scheduled at the beginning. Thus, we need a revised *Initialize* routine and two event routines (Repair and Fail), as listed in Figure 12. By reflecting the event routines of Figure 12 into the single server system main program in Figure 10, we can build a dedicated simulator for the 'single server system with resource failures' given in Figure 11.

2.6 C# Implementation of the Single Server System Event Graph Simulator

This section describes how a dedicated event graph simulator for a single server system is implemented in C#. Figure 13 presents the class diagram for the single server system event graph simulation that consists



(a) Event graph model

No	O.E.	State Change	Edge	Cond.	Action	Delay	D.E.
0	Initialize	Q=0; M=1; Before=0; SumQ=0;	1	True	schedule	0	Arrive
			2	True	schedule	t_f	Fail
1	Arrive	SumQ+=Q*(CLK−Before); Before=CLK; Q++	1	True	schedule	t_a	Arrive
			2	M≡1	schedule	0	Load
2	Load	SumQ+=Q*(CLK−Before); Before=CLK; M−−; Q−−;	1	True	schedule	t_s	Unload
3	Unload	M++;	1	Q>0	schedule	0	Load
4	Repair	M=1;	1	True	schedule	t_f	Fail
			2	Q>0	schedule	0	Load
5	Fail	M=−1;	1	True	schedule	t_r	Repair
			2	True	cancel	0	Unload
6	Statistics	SumQ+= Q*(CLK−Before); AQL= SumQ/CLK;					

(b) Augmented event transition table

Figure 11: A single server system with resource failure.

Execute-Initialize-routine (Now) {Q= 0; M= 1; Before= 0; SumQ= 0; Schedule-event (Arrive, Now); Schedule-event (Fail, Now + t_f)}
Execute-Repair-event-routine (Now) { M= 1; Schedule-event (Fail, Now + t_f); if (Q>0) Schedule-event (Load, Now); }
Execute-Fail-event-routine (Now) { M= −1; Schedule-event (Repair, Now + t_r); Cancel-event (Unload); }

Figure 12: Initialize routine and additional event routines for the resource failure single serve system.

of three classes: *Simulator*, *EventList*, and *Event* classes. Contained in the ***Simulator*** class are the main program (Run method), an initialize routine (*Execute_Initialize_routine*), three event routines, a statistics routine (*Execute_Statistics_routine*), event-handling functions (*Retrieve_Event, Schedule_Event*), and random variate generators (*Exp, Uni*). The *member variables* in the *Simulator* class include: (1) *state variables* (*M, Q*); (2) simulation *clock* (*CLK*); (3) *statistics variables* (*SumQ*, *Before*, and *AQL*); (4) a *random number variable* (*U*) for generating uniform random numbers that will be used to generate *Exp (m)* and *Uni (a, b)* random variates; and (5) the *event-list variable* (*FEL*). The ***EventList*** class contains methods for manipulating the *future event list* (*FEL*), defined as a member variable of the *Simulator* class. The ***Event*** class is about the *next event* and two properties of *Name* (event name) and *Time* (scheduled event time).

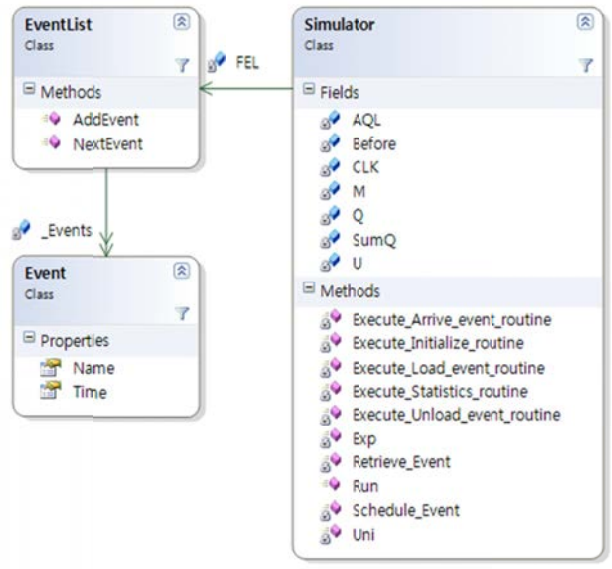


Figure 13: Class diagram for the single server system event graph simulator.

The main program, whose pseudo code is given in Figure 10 (Section 2.5), is implemented using the *Run method* described below. The main program consists of four phases: (1) the *Initialization* phase, (2) the *Time-flow mechanism* phase, (3) the *Event-routine execution* phase, and (4) the *Statistics collection* phase. A complete list of C# codes for the single server system event graph simulator may be found on our website (Choi and Kang 2014).

```
public void Run(double eosTime) {
    //1. Initialization phase
    CLK = 0.0; FEL = new EventList(); U = new Random();
    Event nextEvent = null;
    Execute_Initialize_routine(CLK);
    while (CLK < eosTime) {
    //2. Time-flow mechanism phase
         nextEvent = Retrieve_Event(); CLK = nextEvent.Time;
         //3. Event-routine execution phase
         switch(nextEvent.Name) {
              case "Arrive": { Execute_Arrive_event_routine(CLK);break; }
              case "Load":   { Execute_Load_event_routine(CLK);break; }
              case "Unload": { Execute_Unload_event_routine(CLK);break; } }
    }
    //4. Statistics collection phase
    Execute_Statistics_routine(CLK); }
```

3 HOW TO DEVELOP YOUR OWN DEDICATED ACD SIMULATORS

This section assists with developing your own simulation programs for executing a given ACD model using the *activity scanning algorithm*. The process of developing a dedicated simulator for a given ACD model is described in a *bottom-up manner*, starting from the *primitive functions for handling activities*.

3.1 Functions for Handling Activities

In simulation executions of ACD models, an additional data structure called a *candidate activity list* (*CAL*) is used to handle the *candidate* (or *influenced*) *activities*. A pair of *activity handling functions* is used to store/retrieve activities into/from the CAL, which is a first-in, first-out (FIFO) queue. Figure 14 provides a schematic description of the two *activity handling functions*: Store-activity () and Get-activity (). Initially, there are three candidate activities ({<A1>, <A2>, <A3>}) stored in the CAL. The management of these activities using two functions will be explained with examples.

1. If the Store-activity (A4) function is invoked, the influenced activity (A4) is inserted at the end of the CAL (after <A3>), which is a FIFO queue of influenced activities. Now, the CAL has four *candidate activities*: <A1>, <A2>, <A3>, and <A4>.
2. If the Get-activity () function is invoked, the next activity (<A = A1>) is returned and its entry is removed from the CAL.

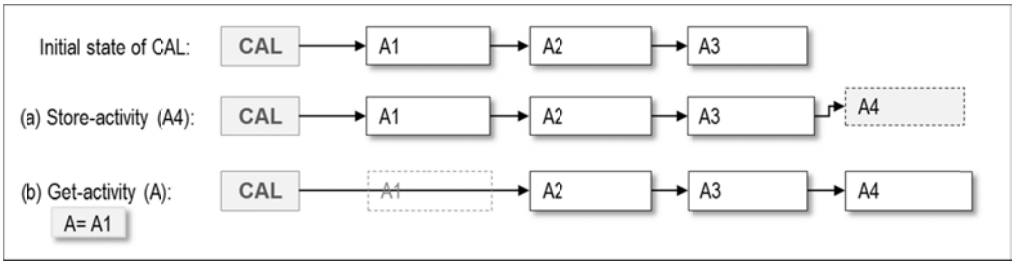
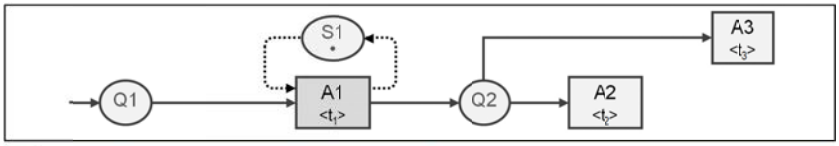


Figure 14: Schematic descriptions of the activity handling functions.

3.2 Activity Routines and Event Routines

Figure 15 displays a portion of an ACD in which (1) Q1 is an input queue of activity A1, (2) Q2 is an output queue of activity A1, and (3) A2 and A3 are influenced activities of activity A1. Queue S1 represents the number of idle resources required to perform activity A1. In the following, the execution rules of an ACD are described for the A1 activity in Figure 15. An activity is confined by two events: an activity-begin event and an activity-end event. Once an *activity-begin event* occurs, the activity-end event is bound to occur after the time delay of the activity duration. Thus, the activity-end event is called a *bound-to-occur event* (BTO event). In the literature, the 'activity-begin' and 'BTO' events are often referred to as 'conditional' and 'bound' events, respectively.



No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	A1	(Q1>0) & (S1>0)	Q1−−; S1−−;	t_1	E1	1	True	S1++;	A1
						2	True	Q2++;	A2, A3

Figure 15: Partial ACD with three activity nodes and three queues.

The At-begin execution rules of activity A1 in Figure 15 are as follows: "If the number of tokens in input queue Q1 is at least one and if there is at least one token in queue S1, then the A1 activity will begin after de-queuing one token from Q1 and one token from S1, and its BTO event E1 is scheduled to occur after the activity duration (t_1)." Similarly, the At-end execution rules are expressed as "one token is created and en-queued into output queue Q2 and a token is returned to queue S1. Then, the influenced activities A2 and A3 are examined for execution". Figure 15 also illustrates an **activity transition table** (for activity A1) that formally specifies the ACD model in a tabular form (Kang and Choi 2010). Specified for each activity node are its At-begin condition and action, BTO-event time and event name, At-end condition and action, and the influenced activity of each output arc.

The abovementioned execution of an activity is separated into the At-begin execution and At-end execution, and is performed using two routines, an activity routine and an event routine, respectively. 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 E1 into the FEL. An activity routine is required for each activity in the activity transition table and it has the following structure: (1) check the At-begin condition and (2) execute the At-begin action and schedule the BTO event of the activity if the at-begin condition is satisfied. The activity routine for activity A1 in Figure 15 can be expressed as follows:

Execute-A1-activity-routine (t) { if ((Q1>0) & (S1>0)) {Q1--; S1--; Schedule-event (EventA1, t)} }.

An *event routine* for ACD is a subprogram that describes the changes in the state variables made at the end of an activity and storing the influenced activities into the CAL. One event routine is required for each activity in the activity transition table and it 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 by invoking the activity handling method *Store-Activity ()*. The event routine for activity A1 in Figure 15 can be expressed as follows:

Execute-E1-event-routine (t) { S1++; Store-activity (A1); Q2++; Store-activity (A2); Store-activity (A3) }.

The event routine of the ACD is similar to that of the event graph in that the changes in the state variables are described. However, the event routine of the ACD stores the influenced activities in the CAL instead of scheduling or canceling future events in the FEL. In an ACD, the next-event scheduling is made during the activity routine.

3.3 Activity Scanning Algorithm

It is described on p. 5 of Hollocks (2008) that the core idea of the *Tocher's three-phase process* came to him at Christmas, 1957, evidently while in his bath! The notion began from the concept of a system that consists of individual *components* progressing as time unfolds through *states* that only change at discrete events. The ***three-phase process*** includes (1) Phase A: advance the *clock* to the time of the next *bound-to-occur (BTO) event*, (2) Phase B: execute the BTO event, and (3) Phase C: initiate 'conditional' activities that the conditions in the model now permit. This three-phase process is formally expressed in an *activity scanning algorithm* as illustrated in Figure 16. The activity scanning algorithm maintains a simulation *clock* (CLK), structure *future event list (FEL)*, and the two *event-handling functions* (Schedule-event () and Retrieve-event ()) that were introduced in Section 2.1.

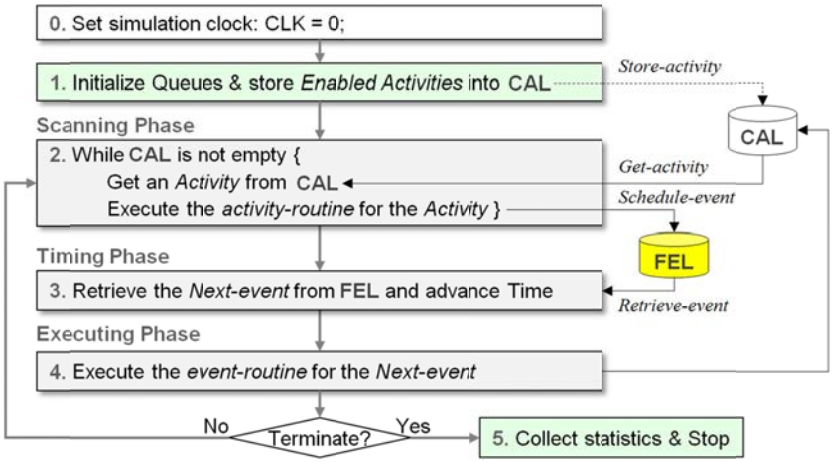


Figure 16: Activity scanning algorithm.

Also, the activity scanning algorithm uses the *candidate activity list* (CAL) and the two *activity handling functions* (Store-activity () and Get-activity ()) that were introduced in Section 3.1. The CAL that stores the influenced activities assists in Phase C of the three-phase process through reducing the number of activities to scan for the execution. Another, perhaps more critical, benefit of introducing CAL is that it allows the management of ***tie-breaking*** among the concurrent activities. Note that the three-phase process is implemented with the sequence of *Phase C* → *Phase A* → *Phase B* in the activity scanning algorithm.

Figure 17 presents the *main program* template of the ACD simulator that implements the activity scanning algorithm. Along with the event graph simulator, the *Initialize* box (Step 1) and *Output statistics* box (Step 5) in Figure 16 are implemented as an *initialize routine* and *statistics routine*, respectively. The simulation is performed inside the *do-while* loop. The *activity routines* for activities A1~An are listed in the *Activity-routines list*. Also, the *event routines* for BTO events E1~En are listed in the *Event-routines list*. Here, the EOS condition is specified using the simulation clock (CLK) and EOS time (t_e). The activity scanning algorithm for *parameterized ACD* is almost the same as that of Figure 16 (Choi and Kang 2013).

```
Begin
    CLK= 0;
    Execute-initialize-routine(CLK);        // (1) Initialize
    do {
        while (CAL is not empty) {          // (2) Scanning phase
            ACTIVITY = Get-activity();
            Case ACTIVITY of {
                A1: Execute-A1-activity-routine (CLK);
                A2: Execute-A2-activity-routine (CLK);
                ...
                An: Execute-An-activity-routine (CLK);
            } // end-of-case                          } Activity-routines list
        } //end-of-while
        Retrieve-event (EVENT, TIME);       // (3) Timing phase
        CLK = TIME;
        Case EVENT of {                     // (4) Executing phase
            E1: Execute-E1-event-routine (CLK);
            E2: Execute-E2-event-routine (CLK);
            ...
            En: Execute-En-event-routine (CLK);
        } // end-of-case                          } Event-routines list
    } while (CLK < te);
    Execute-statistics-routine(CLK);        // (5) Output statistics
End
```

Figure 17: Main program template implementing the activity scanning algorithm.

3.4 Single Server System ACD Simulator

The *process of programming a dedicated simulator* for an ACD model is similar to that for the event graph model presented in Section 2. That is, the overall steps are (1) the (pure) ACD model is converted to an *augmented ACD model* through adding the statistics variables and a statistics routine; (2) an activity transition table is constructed from the augmented ACD model; (3) the initialize routine, event and activity routines, and statistics routine are developed; and (4) the main program is obtained from the main program template given in Figure 17. The process of developing your own ACD simulator is described using the *single server system* model considered in Section 1. Along with the event graph simulator, it is assumed that the EOS time (t_e) is 500 and the distributions of the *inter-arrival times* (t_a) and *service times* (t_s) are *Exp (5)* and *Uni (4, 6)*, respectively

Figure 18 presents an *augmented ACD* for the ACD model described in Figure 2 in order to collect the *average queue length* (AQL) statistics. As mentioned in Section 2.5, two *statistics variables* (SumQ and Before) are introduced. Then, the At-end action of activity Create and At-begin action of activity Process are modified in order to collect the queue length change times. An *activity transition table* for this ACD model is given in Table 2. The Initialize row of the table provides the initial marking and a list of enabled activities, of which the conditions permit initially; the Statistics row lists the expressions for collecting output statistics.

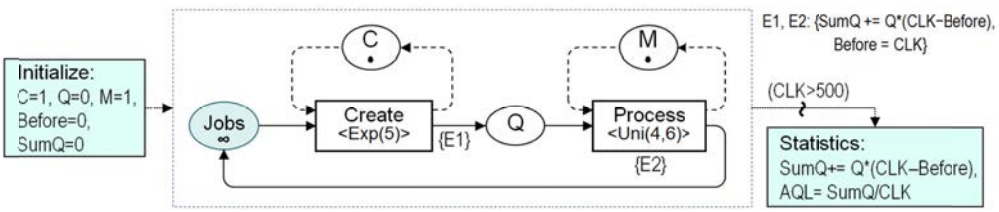


Figure 18: Augmented ACD model for collecting AQL statistics.

Table 2: Activity transition table for the ACD model presented in Figure 18.

No	Activity	At-begin		BTO-event		At-end			
		Condition	Action	Time	Name	Arc	Condition	Action	Influenced Activity
1	Create	(C>0)	C−−;	Exp(5)	Created	1	True	C++;	Create
						2	True	SumQ+=Q*(CLK-Before); Before=CLK; Q++;	Process
2	Process	(M>0) & (Q>0)	SumQ+=Q*(CLK-Before); Before=CLK; M−−; Q−−;	Uni(4,6)	Processed	1	True	M++;	Process
Initalize		*Initial Marking* = {C=1, M=1, Q=0}; *Enabled Activities* = {Create}; *Statistics Variables* = {SumQ=Before=0}							
Statistics		SumQ+=Q*(CLK-Before); AQL=SumQ/CLK;							

The initialize routine, activity and event routines, and statistics routine of the augmented ACD model for collecting the AQL statistics are given in Figure 19. Then, from the main program template of the ACD simulator described in Figure 17, a *single server system ACD simulator* is obtained as described in Figure 20.

```
Execute-initialize-routine (Now) { C = 1; M = 1; Q = 0; SumQ = 0 ; Before = 0; Store-activity(Create); }
Execute-Create-activity-routine (Now) { if (C > 0) { C−−; Schedule-event (Created, Now + Exp(5)); } }
Execute-Process-activity-routine (Now) { if ((M>0) & (Q>0)) { SumQ+=Q*(Now − Before); Before= Now;
  M−−; Q−−; Schedule-event (Processed, Now + Uni(4,6));} }
Execute-Created-event-routine (Now) { if (True) { C++; Store-activity (Create); }
  if (True) { SumQ+=Q*(Now − Before); Before=Now; Q++; Store-activity (Process); } }
Execute-Processed-event-routine (Now) { if (True) { M++; Store-activity (Process); } }
Execute-statistics-routine (Now) { SumQ += Q*(Now – Before); AQL = SumQ / Now; }
```

Figure 19: Routines for the ACD simulator of the single server system.

```
Begin
    CLK= 0;
    Execute-initialize-routine(CLK);
    do {
        while (CAL is not empty) {
            ACTIVITY = Get-activity();
            Case ACTIVITY of {
                Create:   Execute-Create-activity-routine (CLK);
                Process: Execute-Process-activity-routine (CLK);
            } // end-of-case
        } //end-of-while
        Retrieve-event (EVENT, TIME);
        CLK = TIME;
        Case EVENT of {
            Created:     Execute-Created-event-routine (CLK);
            Processed:  Execute-Processed-event-routine (CLK);
        } // end-of-case
    } while (CLK < 500);
    Execute-statistics-routine(CLK);
End
```

Figure 20: Main program of single server system ACD simulator.

3.5 C# Implementation of the Single Server System ACD Simulator

This section describes how a dedicated ACD simulator for the single server system is implemented in C# based on the pseudo codes given in Section 3.4. The single server ACD simulator consists of five classes:

Simulator, *EventList*, *Event*, *ActivityList*, and *Activity* classes. The *Simulator* class contain the main program (Run method), two activity routines (*Create* and *Process*), two event routines (*Created* and *Processed*), event handling functions, activity handling functions, and random variate generators (*Exp* and *Uni*). The *member variables* in the *Simulator* class include: (1) *state variables* (*C*, *M*, and *Q*); (2) *simulation clock* variable (*Clock*); (3) *statistics variables* (*SumQ*, *Before*, and *AQL*); (4) a *random number variable* (*U*) that generates uniform random numbers that are used in generating *Exp (m)* and *Uni (a, b)* random variates; (5) the *event-list variable FEL*; and (6) the *activity-list variable CAL*.

Along with the event graph simulation in Section 2.6, the *EventList class* is defined as a member variable of the *Simulator* class. 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*).

The main program, whose pseudo-code was given in Figure 20, 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. A complete list of C# codes for the single server system ACD simulator may be found in the website (Choi and Kang 2014).

```
public void Run(double eosTime) {
    //1. Initialization Phase
    Clock = 0; FEL = new EventList(); R = new Random();
    CAL = new ActivityList(); Event nextEvent = null;
    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;}}}
       //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;}}
    } while (Clock < eosTime);
    //5. Statistics collection phase
    Execute_Statistics_routine(CLK); }
```

4 A GENERAL PURPOSE ACD EXECUTOR: ACE®

ACE® is the only tool that can execute ACD models. The advantage of ACE® is the use of a formal model as its input (in the form of an *activity transition table*). Figure 21 presents the main window of ACE® (with the single server ACD model from Table 2), which has three main regions: *Main Menu*, *Activity Transition Table (ATT) Window*, and *Spreadsheet Window*. Three tool bars are also provided: *ATT Tool Bar* in the *ATT Window*, and *Queue Tool Bar* and *Variable Tool Bar* in the *Spreadsheet Window*. The ACE® *Menu Bar* contains four menus including *File*, *Model*, *Run*, and *Help*. The *Activity Transition Table (ATT) Window* is where the activity transition table of the ACD model is constructed. The *Spreadsheet Window* is used to declare the queues and variables that appear in the ACD model.

Figure 22 displays the components of ACE®, which consist of three *GUI components* (ATT Editor, Run Options Editor, and Output Report Viewer) and two *library components* (ATT Simulator and Output Report Generator). The GUI components were developed for the model implementation and experimentation. The model implementation can be undertaken using the ATT editor to construct an activity transition table, which is stored in the ATT model, and it consists of a set of queues, a set of

variables, and a set of activity transitions. Prior to the experimentation, the run options should be set to specify the end of simulation (EOS) time, random number seed, and other options for data collection in the Run Options editor. Once the simulation is run successfully, the output report including the system trajectories and statistics with regard to the resources and queues are generated and can be accessed in the Output Report viewer.

The library components are developed in order to simulate the ATT model and generate the output report. The ATT simulator implements the activity scanning algorithm presented in Figure 16, which consists of member variables of CLK, FEL, and CAL, as well as Queues, activity routines, event routines, and the main program. The activity/event routines and main program of the ATT simulator are automatically constructed from the given ATT model with the specified run options before the simulation begins.

The output data is collected using the Publish-Subscribe mechanism of the observer software design pattern (Gamma 1994). Whenever changes in the system states are made, the simulation events are published to the simulator, and then these simulation events are distributed to their subscribers and observers. The observer collects these simulation events and stores them in the collected output data. After the simulation ends, the Output Report Generator transforms the collected output data into an output report data that consists of key performance measures with system trajectories. More detailed information regarding ACE® can be found on our website (Choi and Kang 2014).

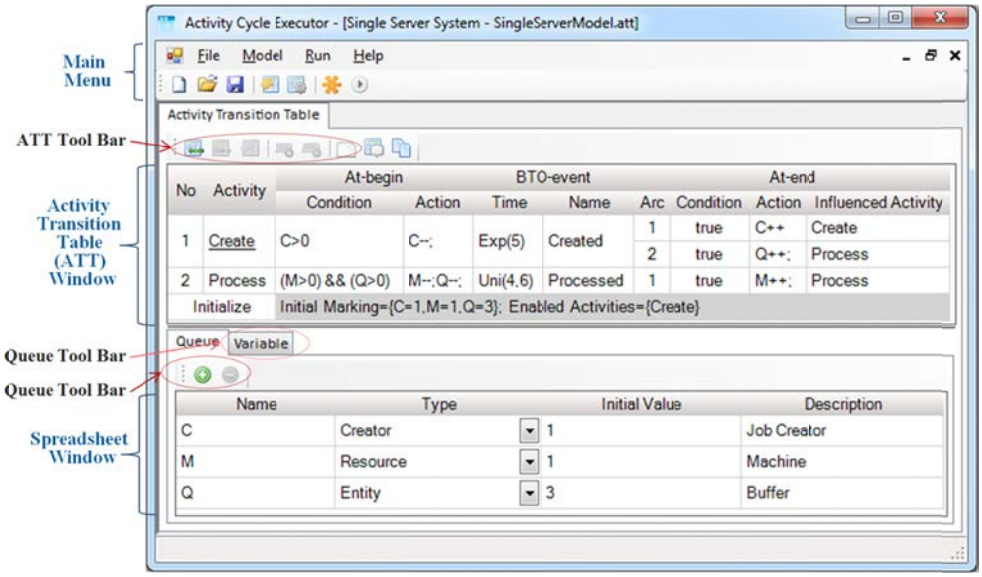


Figure 21: Main window of ACE® with the single server ACD model in Table 2.

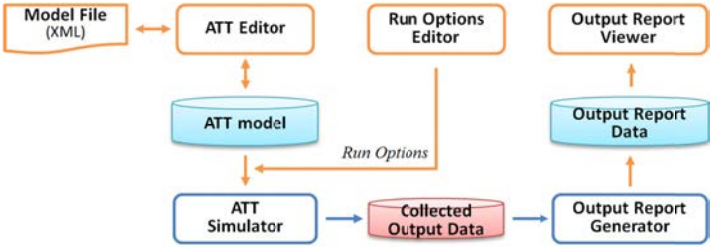


Figure 22: Components of ACE®.

5 SUMMARY

In this tutorial, we have explained how to develop dedicated simulators for executing event graph models and activity cycle diagram (ACD) models. The *event-graph simulator template* was developed from the well-known next-event scheduling algorithm, and the *ACD simulator template* was developed based on the activity scanning algorithm. C# implementations of the simulator templates are presented for a single-server event system. Furthermore, a brief description of a general purpose simulator for executing ACD models is presented.

ACKNOWLEDGMENTS

The tutorial was supported in part by VMS-Solutions Co., Ltd., for which the authors are grateful.

REFERENCES

Carrie, A. 1988. *Simulation of Manufacturing Systems*. John Wiley & Sons.

Choi, B. K., and D. Kang. 2013. *Modeling and Simulation of Discrete Event Systems*. John Wiley & Sons.

Law, A.M. 2007. *Simulation Modeling and Analysis*, 4th edition, McGraw Hill.

Choi, B. K., and D. Kang. 2014. Resources for How to Develop Your Own Simulators for Discrete-Event Systems. Accessed April 8. http://vms-technology.com/book/msdestutorial/.

Choi, B. K, D. Kang, T. Lee, A.A. Jamjoom, and M.F. Abulkhair. 2013. "Parameterized Activity Cycle Diagram and Its Application." *ACM Trans. on Modeling and Computer Simulation* 23(4): Article 24.

Gamma, E., R. Johnson, R. Helm, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hollocks, B. W. 2008. "Intelligence, Innovation and Integrity – KD Tocher and the Dawn of Simulation." *Journal of Simulation* 2(3): 128–137.

Kang, D. H., and B. K. Choi. 2010. "Visual Modeling and Simulation Toolkit for Activity Cycle Diagram." In *Proceedings of the 24th European Conference on Modeling and Simulation*, edited by A. Bargiela, S. A. Ali, D. Crowley, and E. J. H. Kerckhoffs, 169–174. Kuala Lumpur, Malaysia.

Kiviat, P. J., R. Villanueva, and H. M. Markowitz. 1968. *The SIMSCRIPT II Programming Language*, R-460-PR, The RAND Corporation.

Savage, E. L., L. W. Schruben, and E. Yücesan. 2005. "On the Generality of Event-Graph Models." *INFORMS Journal on Computing* 17(1): 3–9.

Schruben, L. W. 1983. "Simulation Modeling with Event Graph Models." *Communications of the ACM* 26(11): 957–963.

Tocher, K. D. 1960. "An Integrated Project for the Design and Appraisal of Mechanized Decision-Making Control Systems." *Operational Research Quarterly* 11(1-2): 50–65.

AUTHOR BIOGRAPHIES

BYOUNG K. CHOI has been a professor in the Department of Industrial and Systems Engineering at KAIST in Daejeon, Republic of Korea, since 1983. He has also been an adjunct professor at King Abdulaziz University in Jeddah, Kingdom of Saudi Arabia, since 2012. He received a Ph.D. in Industrial Engineering from Purdue University in 1982. His current research interests are system modeling and simulation and simulation-based scheduling. His email address is bkchoi@kaist.ac.kr.

DONGHUN KANG is a postdoctoral researcher in the Department of Industrial and Systems Engineering at KAIST in Daejeon, South Korea. He received a Ph.D. from KAIST in Industrial Engineering in 2011. His research interests lie in the DES M&S and its applications in various domains. His email address is donghun.kang@kaist.ac.kr.