Discrete Event Simulation

1 Introduction

1.1 Overview

Discrete event simulation (DES) is a modeling technique used to simulate the behavior and performance of a system as a discrete sequence of events over time. Each event occurs at a specific point in time and marks a change in the state of the system. Key aspects of DES include:

1. **State Transitions (Events):** The fundamental units of the simulation, representing changes in the system's state.
2. **State Variables:** These variables represent the current state of the system. They change in response to events.
3. **Event List:** A list that keeps track of all scheduled events, ordered by their occurrence time. The simulation progresses by processing events in chronological order.
4. **Simulation Clock**: A virtual clock that keeps track of the current time in the simulation. The clock advances to the time of the next event.
5. **Randomness**: Many elements in DES are modeled as random variables to reflect real-world variability, such as arrival times, service times, and failure rates.

1.2 Categories

Based on these definitions, the DES could be categorized into 3 sets:

- **Event Oriented:** In event-oriented simulation, the focus is on the events that change the state of the system. The simulation progresses by processing events in chronological order. It is efficient for systems with events that occur at irregular intervals. The focus is on event scheduling and execution. The key characteristics include:

    1. **Event List**: A list of scheduled events is maintained, typically sorted by time.
    2. **Event Execution**: The simulation clock advances to the time of the next event, and the event is executed.

3. **State Changes**: Events cause changes to the system's state.
4. **Scheduling New Events**: Events may schedule new events to occur at future times.

- **Process Oriented:** Process-oriented simulation focuses on the lifecycle of entities within the system, where each entity follows a specific process or sequence of activities. The key characteristics include:

  1. **Processes:** Each type of entity has a defined process it follows, typically modeled as a sequence of activities or steps.
  2. **Coroutines:** Entities are often implemented as coroutines or threads that simulate the passage of time by yielding control at specific points
  3. **State Changes:** The system state is updated as entities progress through their processes.

- **Activity Scanning**: Activity scanning involves checking the conditions for various activities at regular time intervals, executing any activities whose conditions are met. The key characteristics include:

  1. **Activities and Conditions:** The model consists of a set of activities, each with associated conditions that determine when the activity can be performed.
  2. **Fixed Time Steps:** The simulation clock advances in fixed time steps, and at each step, all activities are scanned to see if their conditions are met.
  3. **State Changes:** If conditions for an activity are met, the activity is executed, and the system state is updated.

2 Department Store Case Study

Suppose there are 3 shopping areas and 3 service desks for payment. When the customers enter the store, they will do the following stuffs:

- Browse and select items in a shopping area.
- Pay at service desk.
- Can browse and select items in another area, or
- Can leave after payment at any desk.

The example of the events for 3 customers is shown below:

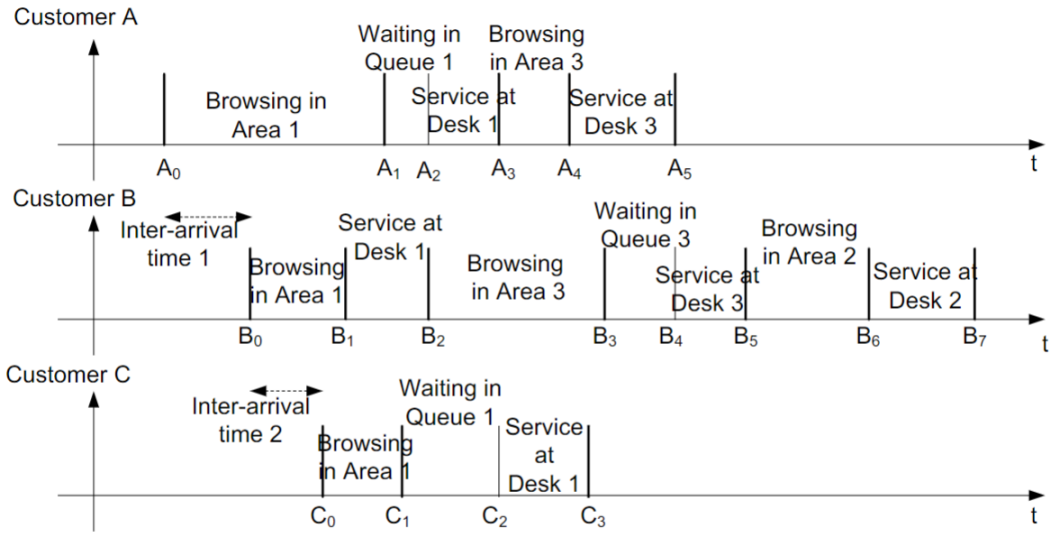

Figure 1: Events Overview

2.1 Event Oriented

If we choose event oriented simulation, then the diagrams will become:

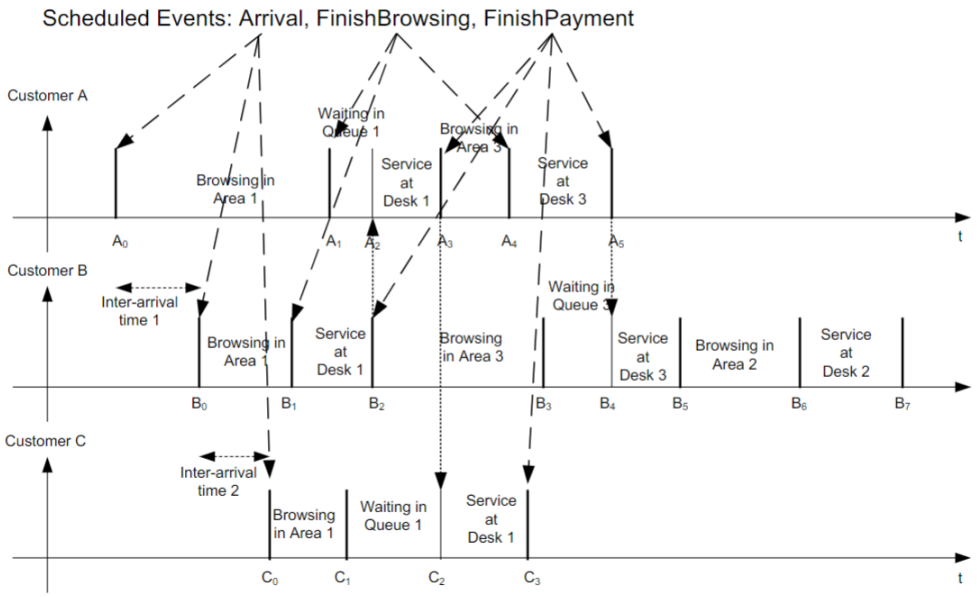

Figure 2: Event Oriented

2.2 Process Oriented

If we choose process oriented simulation, then the diagrams will become:

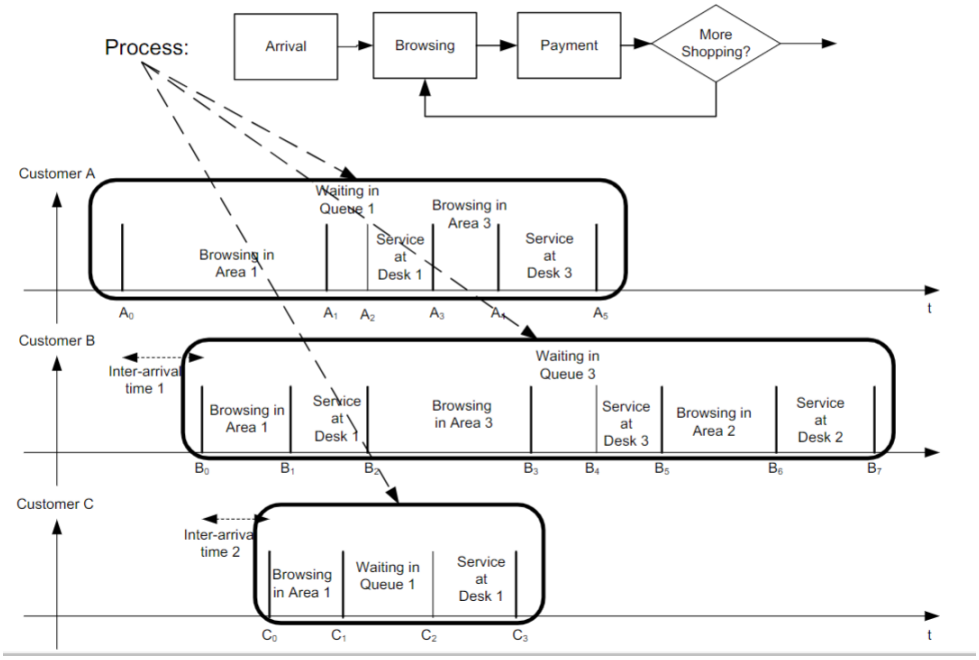

Figure 3: Process Oriented

2.3 Activity Scanning

If we choose activity scanning simulation, then the diagrams will become:

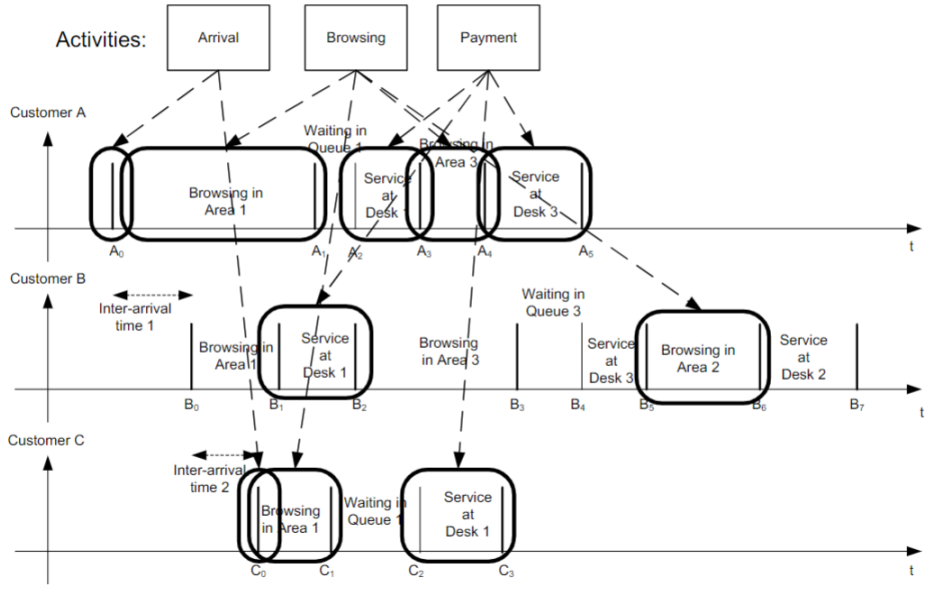

Figure 4: Activity Scanning

3 Airport Traffic Case Study

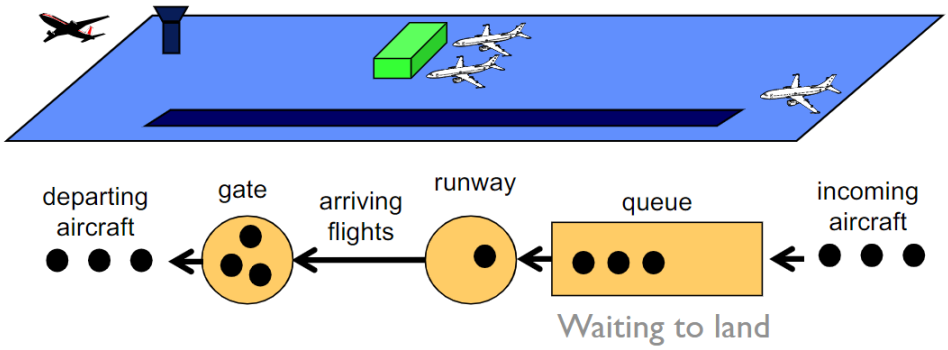

Figure 5: Airport Traffic

Key system behaviors:

- Only one plane at a time can use the runway to land.
- Planes arrive & waiting to land, landing & sit, and then depart
- Once cleared to land, the time to land is some **constant**
- The time on the ground waiting to depart is constant
- Ignore departure queueing

3.1 Event-Oriented

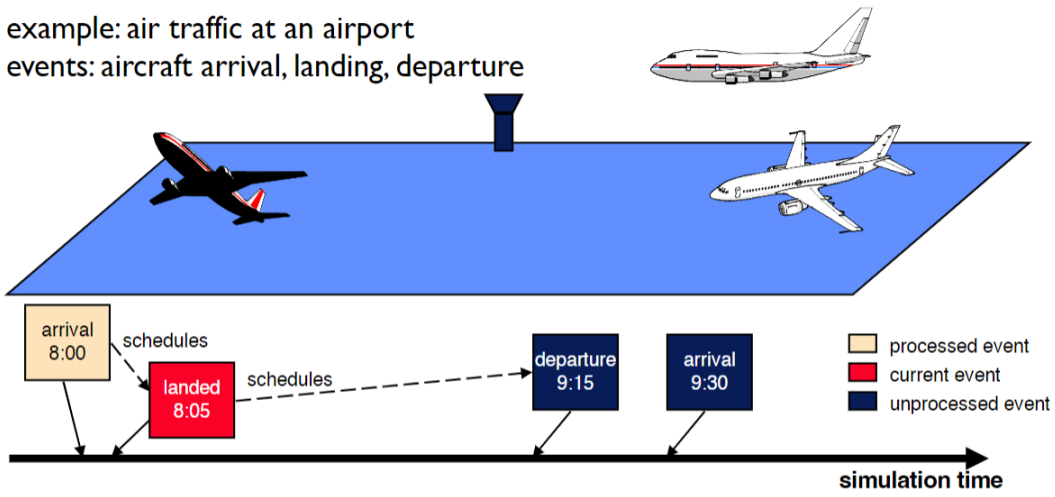

Figure 6: Airport Traffic, Event Oriented

In event-oriented simulation:

- Execution is a sequence of event computations
- Events are processed in timestamp order
- Unprocessed events are stored in a future event list

There are several differences between **Time-stepped** and **Event-driven** simulations. For time-stepped simulations:

1. Discretize time into steps in same step, e.g. $t = 0, 1, 2, 3$
2. Update all state variables concurrently at each step
3. Time advances one-step-at-a-time

And for event-driven simulations:

1. Time can be continuous
2. Only update state variables when events occur
3. Time advances from one event to the next, which may be irregular.

3.2 Event Oriented Implementation

Before the implementation, we need to define several variables. First, the time constants:

- **R**: time runway is used for each landing aircraft (constant)
- **G**: time required on the ground before departing (constant)

State variables:

- **now:** current simulation time
- **in_air**: number of aircraft landing or waiting to land
- **on_ground**: number of landed aircraft
- **runway_free**: Boolean, true if runway available

And the events:

- **arrived**
- **landed**
- **departed**

The interrelation could be shown below:

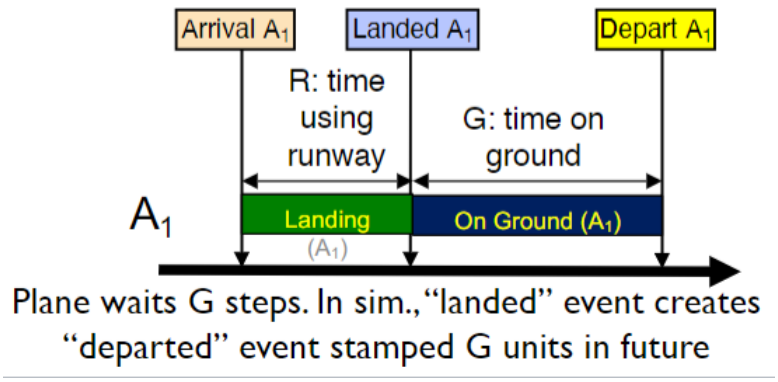

Figure 7: Interrelations

And a possible scenario is shown below:

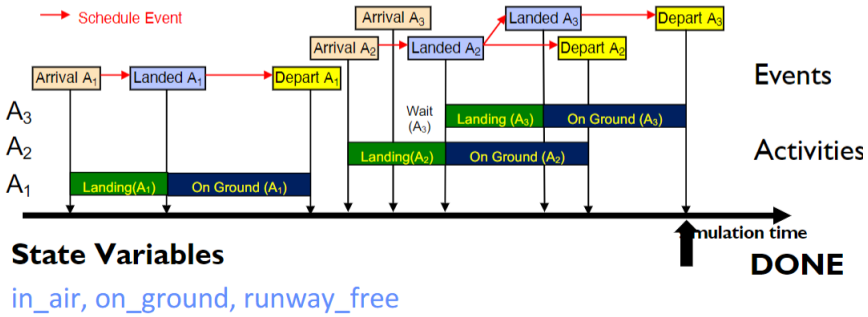

Figure 8: Event Scenario

Now we can start to implement the events:

```
Arrival Event: #New plane is here!
InTheAir := InTheAir+1; #So increment this
if (RunwayFree)
  RunwayFree:=FALSE; #If free, isn't anymore
  Schedule Landed event @ Now + R; #Schedule
```

Figure 9: Arrival Event Implementation

```
Landed Event:
InTheAir:=InTheAir-1; #not in the air anymore
OnTheGround:=OnTheGround+1; #but it is on the ground
Schedule Departure event @ Now + G; #schedule departure
if (InTheAir>0) #someone else can land in R units
   Schedule Landed event @ Now + R;
else #empty runway
   RunwayFree := TRUE;
```

Figure 10: Landing Event Implementation

```
Departure Event:
OnTheGround := OnTheGround - 1; #decrement
```

Figure 11: Departure Event Implementation

A data structure is needed to store events that have been scheduled, but not yet processed, which is called **Future Event List (FEL)**. The required data structure is called a **priority queue**. One possible implementation is shown below:

- Insert (FEL, ev, ts); /* aka enqueue */
  - Add event ev with timestamp ts to event list FEL
- Event = Delete (FEL); /* aka dequeue */
  - Remove smallest time stamp event and return a pointer to it
- DeleteArbitrary (FEL, ev);
  - Unschedule an event
  - Delete an arbitrary event ev (not necessarily smallest timestamped item)

Figure 12: Future Event List Implementation