

Subroutines and Control Abstraction

8

8.6.4 Discrete Event Simulation

EXAMPLE 8.83

Sequential simulation of a complex physical system

Suppose that we wish to experiment with the flow of traffic in a city. A computerized traffic model, if it captures the real world with sufficient accuracy, will allow us to predict the effects of construction projects, accidents, increased traffic due to new development, or changes to the layout of streets. It is difficult (though certainly not impossible) to write such a simulation in a conventional sequential language. We would probably represent each interesting object (automobile, intersection, street segment, etc.) with a data structure. Our main program would then look something like this:

```
while current_time < end_of_simulation
    calculate next time  $t$  at which an interesting interaction will occur
    current_time :=  $t$ 
    update state of objects to reflect the interaction
    record desired statistics
print collected statistics
```



The problem with this approach lies in determining which objects will interact next, and in remembering their state from one interaction to the next. It is in some sense unnatural to represent active objects such as cars with passive data structures, and to make time the active entity in the program. An arguably more attractive approach is to represent each active object with a coroutine, and to let each object keep track of its own state.

If each active object can tell when it will next do something interesting, then we can determine which objects will interact next by keeping the currently inactive coroutines in a priority queue, ordered by the time of their next event. We might begin a one-day traffic simulation by creating a coroutine for each trip to be taken by a car that day, and inserting each coroutine into the priority queue with a “wakeup” time indicating when the trip is to begin:

EXAMPLE 8.84

Initialization of a coroutine-based traffic simulation

```
coroutine trip(...)
...
for each trip t
  p := new trip(...)
  schedule(p, t.start_time)
```

EXAMPLE 8.85

Traversing a street segment
in the traffic simulation

Let us assume that we think of street segments as passive, and represent them with data structures. At any given moment, we can model a segment by the number of cars that it is carrying in each direction. This number in turn will affect the speed at which the cars can safely travel. Whenever it awakens, the coroutine representing a trip examines the next street segment over which it needs to travel. Based on the current load on that segment, it calculates how much time it will take to traverse it, and schedules itself to awaken again at an appropriate point in the future:

```
coroutine trip(origin, destination : location)
  plan a route from origin to destination
  detach
  for each segment of the route
    calculate time i to reach the end of the segment
    schedule(current_coroutine, current_time + i)
```

EXAMPLE 8.86

Scheduling a coroutine for
future execution

The `schedule` operation is easily built on top of transfer:

```
schedule(p : coroutine; t : time)
  -- p may be self or other
  insert (p, t) in priority queue
  if p = current_coroutine -- self
    extract earliest pair (q, s) from priority queue
    current_time := s
    transfer(q)
```

EXAMPLE 8.87

Queueing cars at a traffic
light

In some cases, it may be difficult to determine when to reschedule a given object. Suppose, for example, that we wish to more accurately model the effects of traffic signals at intersections. We might represent each traffic signal with a data structure that records the waiting cars in each direction, and a coroutine that lets cars through as the signal changes color:

```
record controlled_intersection =
  EW_cars, NS_cars : queue of trip
  const per_car_lag_time : time
  -- how long it takes a car to start after its predecessor does
  coroutine signal(EW_duration, NS_duration : time)
    detach
    loop
      change_time := current_time + EW_duration
      while current_time < change_time
        if EW_cars not empty
          schedule(dequeue(EW_cars), current_time)
          schedule(current_coroutine, current_time + per_car_lag_time)
```

```

change_time := current_time + NS_duration
while current_time < change_time
  if NS_cars not empty
    schedule(NS_cars.dequeue(), current_time)
  schedule(current_coroutine, current_time + per_car_lag_time)

```

EXAMPLE 8.88

Waiting at a light

When it reaches the end of a street segment that is controlled by a traffic signal, a trip need not calculate how long it will take to get through the intersection. Rather, it enters itself into the appropriate queue of waiting cars and “goes to sleep,” knowing that the `signal` coroutine will awaken it at some point in the future:

```

coroutine trip(origin, destination : location)
  plan a route from origin to destination
  detach
  for each segment of the route
    calculate time i to reach the end of the segment
    schedule(current_coroutine, current_time + i)
    if end of segment has a traffic light
      identify appropriate queue Q
      Q.enqueue(current_coroutine)
      sleep()

```

EXAMPLE 8.89

Sleeping in anticipation of future execution

Like `schedule`, `sleep` is easily built on top of transfer:

```

sleep()
  extract earliest pair (q, s) from priority queue
  current_time := s
  transfer(q)

```

The `schedule` operation, in fact, is simply:

```

schedule(p : coroutine; t : time)
  insert (p, t) in priority queue
  if p = current_coroutine
    sleep()

```

Obviously this traffic simulation is too simplistic to capture the behavior of cars in a real city, but it illustrates the basic concepts of discrete event simulation. More sophisticated simulations are used in a wide range of application domains, including all branches of engineering, computational biology, physics and cosmology, and even computer design. Multiprocessor simulations (see reference [VF94], for example) are typically divided into a “front end” that simulates the processors and a “back end” that simulates the memory subsystem. Each coroutine in the front end consists of a machine-language interpreter that captures the behavior of one of the system’s microprocessors. Each coroutine in the back end represents a load or a store instruction. Every time a processor performs a load or store, the front

end creates a new coroutine in the back end. Data structures in the back end represent various hardware resources, including caches, buses, network links, message routers, and memory modules. The coroutine for a given load or store checks to see if its location is in the local cache. If not, it must traverse the interconnection network between the processor and memory, competing with other coroutines for access to hardware resources, much as cars in our simple example compete for access to street segments and intersections. The behavior of the back-end system in turn affects the front end, since a processor must wait for a load to complete before it can use the data, and since the rate at which stores can be injected into the back end is limited by the rate at which they propagate to memory.

✓ CHECK YOUR UNDERSTANDING

80. Summarize the computational model of discrete event simulation. Explain the significance of the time-based priority queue.
 81. When building a discrete event simulation, how does one decide which things to model with coroutines, and which to model with data structures?
 82. Are all inactive coroutines guaranteed to be in the priority queue? Explain.
-