Modelling Discrete-Event Systems Using DEVS (2016 Fall)

Assignment1: Kitchen Simulator

Chuan Xiong 4114880

University of Ottawa

Part I : Conceptual model Description

Restaurant simulator simulates the environment in a restaurant's kitchen. The kitchen consists of multiple chiefs and waiters. For simplicity, this simulator will only have two chiefs and one waiter. The waiter takes the orders from customers and delivers the order to the kitchen. Each chief can only cook one order at time, once he finish cooking this order, he will pass the dish to the waiter by notifying the waiter order is done by him. Then he will be available to take the next order.

Initially, both chief are in idling state, once waiter have sent out an order, one of the chief will take this order, and change his states to busy. Once order is done, he sends a message to waiter notifying the order is done, and changes his state to idle. Then becomes ready to cook the next order. If both chief are busy, the waiter will hold on to the remaining order in a waiting list/queue, until one of the chief passes him an order.

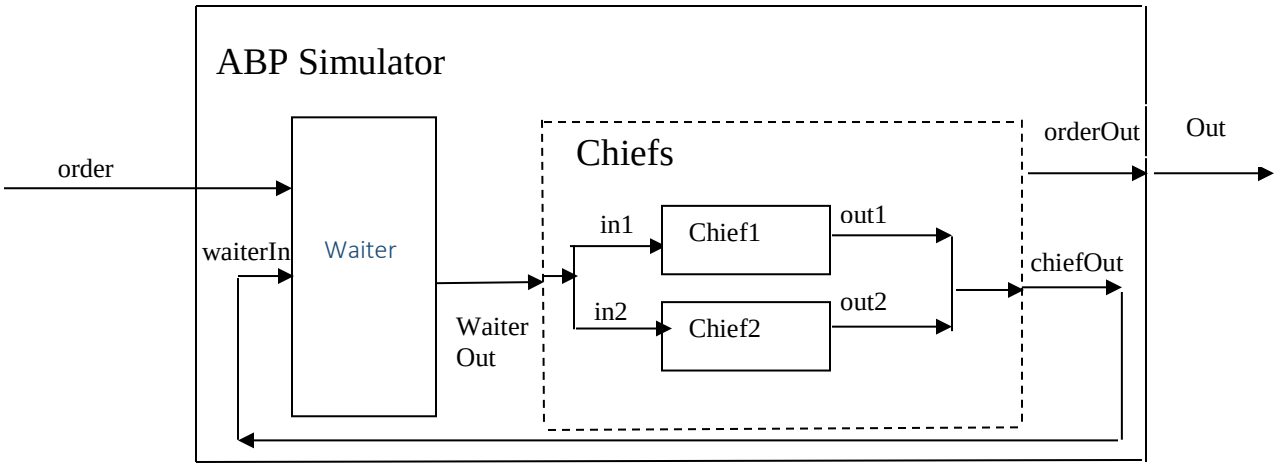


Figure 1 Structure of Kitchen Simulator

The behavior of both chiefs are identical. They are initially in idling states, until waiter send an order to the kitchen with order number. The order will be either one or two, which indicates which chief it is assigning to. Both chief will check the order number, and only one chief will proceed to complete the order. Once completing the order, the chief will pass

the order to waiter through chiefOut with a number *orderbit* indicating which one of the chief have completed this order. At the same time, the chief will send an output to the whole system.

The behavior of the waiter is very much like the waiters in real world. It will have an input *order* as a system input, we can manually assigning input orders to start the simulation process, which acts as customers. Waiter will have another input *waiterIn* that is connected with *chiefOut*. This port's purpose is to get notification from chiefs and update the status of each chief, so waiter will know which chief he can assign the next order to. Waiter initially is in idling states, it will become active when he receives order from customers/*order* port. Both chiefs have initial available status to him, he will always give the first order to chief1 if both chiefs are available. Once an order is passed to chief1, chief1 becomes unavailable to him, and will only become available when *chiefout* sends an *orderbit* 1 to the waiter. Waiter will have a local variable *ordercounter* that keeps track of number of orders he had taken, once he pass an order to a chief, this counter will decrement by one. The waiter becomes passive once all of the orders are completed/sent out.

Part II : Atomic Model and Coupled Model specificaiton

As shown in Figure 1, the Kitchen Simulator has 1 input and 1 output. The *order* input indicates there is an order going to the waiter, which will stimulate the waiter model. The output *orderout* show the time when an order has been complete, and the value of this output shows which chief has completed this order.

Formal Specifications

The formal specifications <S, X, Y, δint, δext, λ, ta> for the atomic models are defined as follows:

Chief1 and Chief2:

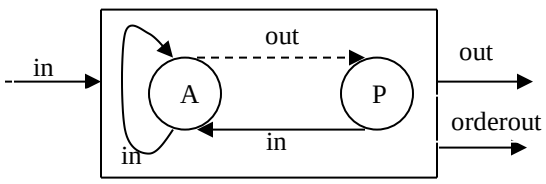
S = {passive, active}

X = {in}

Y = {out, orderout}

δint (active) = passive

δext (in, active) = active



λ(active)

{ send chief's number to port *out //this is to notify waiter*

send chief's number to port order*out //this is system output*

}

ta(passive) = INFINITY

ta(active) = delay (normal distribution) or constant time based on which chief for easy identification

Waiter:

State Variables:

sigma = INFINITY, phase = Passive;

orderbit = 0; //local variable indicating which chief he should send the output to

order_counter = 0 //the total number of orders still need to be done

chief1 = 1; // chief1's status, 1=available, 0=unavailable

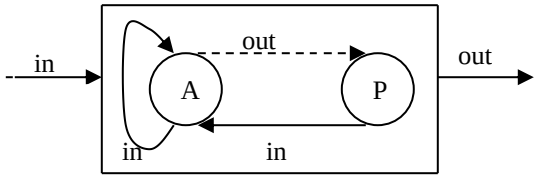
chief2 = 1; // chief2's status

Formal specification:

X = {order, waiterIn}

Y = {waiterOut}

S = {passive, active}



δext (order_counter, order, order_bit)

{ case phase

passive:

active:

if msg is from order

{

order_counter++

//increment number of orders

}

if msg is from waiterIn

if msg == 1 then chief1 =1

if msg == 2 then chief2 =1

//updating chief's status based on the value of the message

//1=available, 0 = unavailable

```
}

δint (order_counter, chief1, chief2, order_bit)
{       case phase
        active:
                if (order_counter less than or equal to 0)  passivate
                //put model to sleep since no more orders at the moment

                elseif (chief1 and chief2 equal to 0)
                {
                        Order_bit = 0 // this will result in output function not sending
                        any output as both chiefs are busy
                }
                elseif (chief1 equal 1 and order_counter >0) //send order to chief1
                {
                        Order_bit = 1
                        phase = active;
                        sigma = preparationtime;
                }

        elseif (chief2 equal 1 and order_counter >0) //send order to chief2
        {
                        Order_bit = 2
                        phase = active;
                        sigma = preparationtime;
        }
        passive: /*Never happens*/
}

λ(active)
{
if (order_bit equal 1) send order_bit to ChiefIn
elseif(order_bit equal 2) send order_bit to ChiefIn
}
```

The formal specifications <X, Y, D, {M_i}, {I_i}, {Z_{ij}}, SELECT > for the coupled model Chiefs and Kitchen Simulator are defined as follows:

Chiefs:

X = {chiefIn};
Y = {chiefOut};
D = {chief1, chief2};
I(chief1) = self;
I(chief2) = self;
Z(chief1) = self;
Z(chief2) = self;
SELECT: ({chief1, chief2}) = chief1;

Kitchen Simulator:

X = {order};
Y = {orderout};
D = {waiter, Chiefs};
I(waiter) = {self};
I(Chiefs) = {self};
Z(waiter) = Chiefs; Z(waiter) = self;
Z(Chiefs) = waiter; Z(Chiefs) = receiver;
SELECT: ({waiter, Chiefs}) = Waiter;

Test Strategies

The atomic models and coupled models will be tested using the black box testing method. Test cases are created by adding different combinations of inputs to the event file (*.ev*), run the simulation and check whether the outputs in the output file (*.out*) are what we expected.

Part III: Test Cases and Execution Analysis

In order to verify the atomic models and coupled models, test cases are created to test these models.

Atomic Model chief1 and chief2:

Both chief1 and chief2 have almost identical implementation beside the *order_bit* they will receive from the external input function. To test this model, we are assuming that we will be receiving input one at a time with gaps in between each input, since the waiter will mark this chief as unavailable after sending him one order, and mark him as available after receiving an output from this chief. The *chief1.ev* file is created as following:

00:00:00:00 in 1
00:00:05:00 in 1
00:00:10:00 in 2
00:00:15:00 in 1
00:00:20:00 in 1

The events with **bold** fonts should not generate any outputs because the order_bit does not match with the chief number, but in theory chief1 will never receive an order that does not belong to him as the waiter model's implementation will handle such case. The output file *chief1.out* shows the expected results

00:00:02:987 out 1
00:00:06:796 out 1
00:00:18:035 out 1
00:00:22:182 out 1

Atomic Model waiter:

The input value to port *order* is not relevant, since we will take each input to port *order* as one input. The *waiterIn* port's value (must be 1 or 2 in order for the simulation to continue) indicates which chief has send an output.

The *waiter.ev* file is created as follows.

00:00:00:00 order 3
00:00:00:00 order 33
00:00:00:00 order 3
00:00:00:00 order 3
00:00:00:00 order 3
00:00:00:00 order 3
00:00:15:00 waiterIn 1
00:00:16:00 waiterIn 2
00:00:25:00 waiterIn 1

00:00:26:00 order 1
00:00:28:00 waiterIn 2
00:00:30:00 waiterIn 2
00:00:38:00 waiterIn 1

Both chiefs are initially available to the waiter, then the waiter receives input from the chief at time 15 indicating they have become available, so between finishing up the first two orders and time 15, nothing was sent from waiter. The number of output matches with the number of orders.

00:00:02:000 waiterout 1
00:00:03:000 waiterout 2
00:00:17:000 waiterout 2
00:00:18:000 waiterout 1
00:00:27:000 waiterout 1
00:00:29:000 waiterout 2
00:00:31:000 waiterout 2

Coupled Model Chiefs:

The coupled model Network consists of two chiefs' models. The two chiefs work independently. The test result is similar to that of the atomic model chief. The *Chiefs.ev* is created as follows:

00:00:00:00 chiefIn 1
00:00:00:00 chiefIn 2
00:00:05:00 chiefIn 1
00:00:06:00 chiefIn 2
00:00:10:00 chiefIn 1
00:00:11:00 chiefIn 2
00:00:15:00 chiefIn 1
00:00:16:00 chiefIn 2
00:00:21:00 chiefIn 1

chief1 we are using a constant time for processing, chief2 we are using a random function. Once again, the input can only be 1 or 2, and we are receiving the correct order of outputs, *chiefout* is the input value to *waiterIn* as previously discussed.

00:00:01:000 chiefout 1

00:00:01:000 orderout 1
00:00:02:987 chiefout 2
00:00:02:987 orderout 2
00:00:06:000 chiefout 1
00:00:06:000 orderout 1
00:00:07:796 chiefout 2
00:00:07:796 orderout 2
00:00:11:000 chiefout 1
00:00:11:000 orderout 1
00:00:12:957 chiefout 2
00:00:12:957 orderout 2
00:00:16:000 chiefout 1
00:00:16:000 orderout 1
00:00:19:035 chiefout 2
00:00:19:035 orderout 2
00:00:22:000 chiefout 1
00:00:22:000 orderout 1

Coupled Model Kitchen Simulator:

The coupled model Kitchen Simulator is the top model which integrates atomic models waiter and coupled Chiefs. The will be one input *order* and one output *orderout*

The following is an example of the *kitchen.out.*

00:00:00:00 order 1
00:00:00:00 order 5
00:00:00:00 order 2
00:00:54:00 order 2
00:00:54:00 order 2
00:00:54:00 order 2
00:00:54:00 order 2
00:00:54:00 order 5

As we can see, 3 order inputs are given at time 0,and we have 3 outputs received around that time; then we have another 5 inputs received at time 54, so we have 5 more outputs received after time 54.

00:00:03:000 orderout 1
00:00:05:000 orderout 1
00:00:05:987 orderout 2
00:00:57:000 orderout 1
00:00:58:796 orderout 2
00:00:59:000 orderout 1
00:01:01:000 orderout 1
00:01:02:957 orderout 2

The Kitchen Simulator model simulates the Kitchen environment and generates the expected results. The behaviors and features of the waiter and Kitchen are simulated by the respective models. The testing cases verify the specifications of models. The Kitchen Simulator works exactly as we expected according to the specifications.