12 SimJava

As discussed in the previous lecture note, process based simulation focuses primarily on entities within the system, rather than events. The model is developed as the series of interactions between these entities. In the language we will consider, interactions are represented by messages; in other languages other mechanisms are used to capture interactions. But in all process based simulations the key elements of any model are the entities and their interactions.

SimJava2[1] is a Java API intended to aid in the development of discrete event simulation models. SimJava2 augments Java with building blocks which are designed to be useful in modelling a wide range of systems. In particular, the package includes a class definition `Sim_entity` and all entities within the system are represented by subclasses of this class. It is assumed that entities each have their own pattern of behaviour which is why the general class `Sim_entity` needs to be specialised for each case.

The SimJava2 web pages (`http://www.dcs.ed.ac.uk/home/simjava`) provide documentation and a tutorial for SimJava2 as well as several example models. You will also find examples on the module web page. In this course you will be expected to be able to understand, modify and write SimJava2 programs, and to generate, understand and interpret the output from such programs. Thus you need to develop a basic familiarity with the language.

12.1 Example: Variations on a Queue

In this section we consider several modified versions of the deterministic queue presented at the end of the previous lecture note. The objective is to introduce various features of SimJava2. In a later lecture note we will consider a more sizeable model. First we go through the deterministic queue in a little more detail, explaining the elements of the `simjava` package.

```
import eduni.simjava.*;

class Source extends Sim_entity {
  private Sim_port enqueue;

  public Source(String name) {
    super(name);
    enqueue = new Sim_port("enqueue");
    add_port(enqueue);
  }

  public void body() {

    while (Sim_system.running()) {
      sim_schedule(enqueue,0.0,0);
```

[1]SimJava2 is the latest release of SimJava which has much improved statistical support. This is the version we will be using in this course.

```
    sim_pause(10.0);
  } } }
```

All SimJava2 models start with the line: `import eduni.simjava.*;` which imports all the classes of the `simjava` package. Each entity in the model is a subclass of the provided class `Sim_entity`; this includes both entities representing "real things" in the system being modelled and modelling entities which are included to control the system. In this simple queueing system there are just two entities: the arrival process which generates customers and the server which handles service requests. In this model the "queue" in the queueing system is modelled implicitly by an internal queue of the simulation.

The arrival process is modelled as a subclass of `Sim_entity` named `Source`. Each subclass of `Sim_entity` must have a constructor which calls the constructor for `Sim_entity`, passing the string which is the identifier for this instance as a parameter. The rest of the constructor is used to set up the attributes (ports, distributions, local parameters) of the entity. In this case there is just a single port which will be used to communicate with the server. As discussed in the last lecture note the arrival of a customer is represented by an event, generating a message from the arrival process to the server. The port `enqueue` is first instantiated and then added to the entity.

The second part of the entity declaration is the `body` method. This is called when the simulation is run, and details the behaviour of the entity. The behaviour of the entity is controlled by a while loop conditioned on a call to the boolean function `Sim_system.running()`. As long as the condition is true the behaviour will be repeated, i.e. another customer will be generated. On each iteration a customer is generated, represented by an event, dispatched from the `enqueue` port. The generation of an event is scheduled (using `sim_schedule`) to occur after *delay* 0.0, and is given a *tag* value 0 in this case. Tags are integer values that can be used to distinguish different types of events. This can be useful if an entity may receive events from several sources and needs to distinguish, for example, the order in which they are handled. The method `sim_pause(10.0)` blocks the entity for 10.0 simulation time units; implicitly this removes the entity from the head of the event list and reschedules it for the current time + 10.0.

```
class Server extends Sim_entity {
  private Sim_port arrival;

  public Server(String name) {
    super(name);
    arrival = new Sim_port("arrival");
    add_port(arrival);
  }

  public void body() {
    Sim_event next = new Sim_event();

    while(Sim_system.running()) {
      sim_wait(next);
      sim_process(8.5);
```

```
    sim_completed(next);
  } } }
```

The server entity is declared analogously. Instead of generating arrival events, it waits for such events: `sim_wait`. In this case we are not interested in any specifics of the event, since all events are customer arrivals, but in general, having obtained the event `next` we can then access its attributes using accessor methods such as `next.get_tag()` and `next.event_time()`. As in the source the server entity has a delay, in this case `sim_process(8.5)` rather than `sim_pause` indicating that the entity is busy during the delay rather than idle. When service is complete `Sim_system` is notified that the processing of the event is finished using the call `sim_completed(next)`.

```
class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source customers = new Source("Customers");
    Server service = new Server("Service");
    Sim_system.link_ports("Customers", "enqueue", "Service", "arrival");
    Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                          "Service", 0, 10, False);
    Sim_system.run();
  } }
```

The final part of the program, is the actual "model": here entities are instantiated and ports are linked together, controlling which entities can communicate[2]. This takes place in a class declaration; this class declaration shares the name of the file and has a `main` method. First, `Sim_system` object is initialised. This is followed by the entity instantiations. Port connections are uni-directional single links, and are specified by *Sending Object Name, Sending Port Name, Receiving Object Name, Receiving Port Name*. The *termination* or *stopping* condition for the simulation needs to be set[3]. Termination conditions can be defined in terms of

- the number of event completions which have been observed in the model;
- the length of (simulation) time for which the model has been running; or
- the estimated accuracy of the measures being collected (see Lecture 13).

In this case we stipulate that the simulation should run until 10 service events have been completed. Since the simulation engine cannot judge when an event can be judged to be complete the modeller is responsible for marking when events are completed (see the statement `sim_completed(next);` at the end of the body of the `Server` entity). Finally the call `Sim_system.run()` starts the simulation; this method will exit when there are no more events to process. Each entity will cease to generate events when the termination condition is satisfied.

[2] Not strictly true since events can be sent outside the port-based messaging system, for example in animated simulations when you do not want an explicit link in the display.

[3] If this is not set the simulation will run indefinitely until its process is killed by the user or it crashes. When animation is being used it can make sense to leave the control of the termination of the simulation to the user who can make a decision based on what he/she observes on the screen.

```
class Source extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj src_hold;

  public Source(String name) {
      .........
    src_hold = new Sim_negexp_obj("iatime", 10.0);
  }

  public void body() {
    while (Sim_system.running()) {
      sim_schedule(enqueue,0.0,0);
      sim_hold(src_hold.sample());
    } } }

class Server extends Sim_entity {
  private Sim_port arrival;
  private Sim_negexp_obj svr_hold;

  public Server(String name) {
      .........
    svr_hold = new Sim_negexp_obj("stime", 8.5);
  }

  public void body() {
    Sim_event next = new Sim_event();
    while(Sim_system.running()) {
      i++; if(i>10) break;
      if (sim_waiting() > 0)
        sim_select(Sim_system.SIM_ANY, next);
      else
        sim_wait(next);
      sim_process(svr_hold.sample());
      sim_completed(next);
    } } }
```

Figure 25: Source and Server declarations for a Markovian queue

12.1.1 Using random variables

Since we are primarily interested in stochastic systems in this course the first generalisation we make of the simple queueing system is to change the interarrival time (the `sim_pause()` at the end of the body of `Source`) and the service time (the `sim_process()` at the end of the body of `Server`). To do this we define appropriate random variables in each constructor. In the source we define an exponentially distributed random variable `src_hold`. This is a private attribute of the entity and is instantiated in the constructor:

```
src_hold = new Sim_negexp_obj("iatime",10.0);
```

The first parameter is an identifier of the random variable, the second is the *mean* of the exponential distribution. Note that the value used to specify the exponential distribution is its *mean*, **not** its *parameter*, e.g. the distribution $F(t) = 1 - e^{rt}$ would have an instantiation like:

```
new Sim_negexp_obj("iatime",1/r)
```

In the modified model, each time the source delays between arrivals instead of the deterministic value 10.0 it takes an exponentially distributed amount of time with mean 10.0. Sampling a value for the random variable is generated by the method call `scr_hold.sample()`. The declaration and use of `svr_hold` in the server is analogous. (see Figure 25.)

Now that the delays are no longer deterministic it cannot be guaranteed that the server is ready to receive each arrival as it is sent from the source. If an event arrives at an entity when it is not ready to handle it, the event is entered into a *deferred queue*. Events can be extracted from the deferred queue using `sim_select(Sim_predicate pred, Sim_event ev)`; predicates are used to select specific events from the queue. A method `sim_waiting()` is provided to test the number of events currently in the deferred queue for that entity. In the body of the server, when the server is ready to begin a new service it now first checks to see if any events are waiting in the deferred queue (`if (sim_waiting() > 0)`) and if there are any selects one of them (`sim_select(Sim_system.SIM_ANY, next);`); alternatively it waits for the next event to be sent to it as in the previous version of the model (`sim_wait(next);`). In fact the construction:

```
if (sim_waiting() > 0)
   sim_select(Sim_system.SIM_ANY, ev);
else sim_wait(ev);
```

appears so often that it is implemented as a method `sim_get_next(ev)`

`Sim_system.SIM_ANY` is a wildcard predicate which matches any event. It is also possible to select just the first event that has a given tag using `sim_select(new Sim_type_p(tag), ev)`. There is a corresponding form of `sim_get_next()` which takes a predicate and a tag as arguments.

12.1.2 Reporting

SimJava2 provides support for statistics gathering and reporting of both default measures and custom measures. Default measures can be regarded as common to all SimJava2 entities and events, whatever is actually being modelled. For default measures the modeller does not need to explicitly tell SimJava2 when measurements need to be taken, only which default measures are of interest. Note, however, that it is up to the modeller to ensure that the default measures do make sense in the context of the chosen entity in their model. The Server entity of the Queue model extended with default measures is shown in Figure 26.

```
class Server extends Sim_entity {
  private Sim_port arrival;
  private Sim_negexp_obj svr_hold;
  private Sim_stat stat;

  public Server(String name) {
    super(name);
    arrival = new Sim_port("arrival");
    add_port(arrival);
    svr_hold = new Sim_negexp_obj("stime", 8.5);
    add_generator(svr_hold);
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.UTILISATION);
    stat.add_measure(Sim_stat.SERVICE_TIME);
    stat.add_measure(Sim_stat.QUEUE_LENGTH);
    stat.calc_proportions(Sim_stat.QUEUE_LENGTH, new double[] {0, 1, 2});
    stat.set_efficient(Sim_stat.SERVICE_TIME);
    stat.set_efficient(Sim_stat.QUEUE_LENGTH);
    set_stat(stat);
  }

  public void body() {
    Sim_event next = new Sim_event();
    while (Sim_system.running()) {
      sim_get_next(next);
      sim_process(svr_hold.sample());
      sim_completed(next);
    }
  }
}
```

Figure 26: Server entity from the `Queue_Stats.java` model

In contrast, custom measures are particular to each model, and depend on the meaning of the entities and events. In this case, as well as declaring the measures, the modeller must explicitly tell SimJava2 when to update them. Both custom and default measures can be recorded in detail or efficiently. Recording a measures efficiently means taht less data is stored and this improves the efficiency of the simulation run. However, there is also much less flexibility about how the measure can subsequently be manipulated and displayed. Further details can be found in the SimJava2 tutorial.

```
class Source extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj src_hold;
  private Sim_stat stat;

  public Source(String name) {
    super(name);
    enqueue = new Sim_port("enqueue");
    add_port(enqueue);
    src_hold = new Sim_negexp_obj("iatime", 10.0);
    add_generator(src_hold);
    stat = new Sim_stat();
    stat.add_measure("Generation rate", Sim_stat.RATE_BASED);
    set_stat(stat);
  }

  public void body() {
    while (Sim_system.running()) {
      stat.update("Generation rate", Sim_system.sim_clock());
      sim_schedule(enqueue, 0.0, 0);
      sim_pause(src_hold.sample());
    }
  }
}
```

Figure 27: Source entity from the `Queue_Stats.java` model

All custom measures must be classified as

Rate based measures: these record the occurrences of an event over a period of time and derive a rate for the event accordingly.

State based measures: these reflect the proportion of time an entity may be in a certain state. A state based measure is considered to be *continuous* if all states are of interest, and *non-continuous* if there are some states which can be ignored from a measurement point of view.

Interval based measures: these are concerned with time intervals, usually as experienced by events rather than entities. An example might be the waiting time of an event at an entity.

At the end of each experiment a report file (`sim_report`) is produced with details of the default and custom measures together with general information about the experiment[4], such as the time it took to complete and the conditions used. The report generated by the `Queue_Stats.java` model is shown in Figures 28 and 29 .

[4]How an experiment can be constructed from a number of model runs will be explained in Lecture note 13.

```
############################################################
#                                                          #
#                   SIMULATION REPORT                      #
#                                                          #
############################################################

Version: SimJava 2.0

Simulation date:         November 12, 2002
Simulation start time:   11:25:18 AM GMT
Simulation end time:     11:25:18 AM GMT

############################################################
#            Overall simulation run information            #
############################################################

Total simulated time:    107.52726357025585
Total transient time:    0.0
Total steady state time: 107.52726357025585
Transient condition:     None
Termination condition:   10 event completions at Service
Output analysis method:  None
```

Figure 28: Report generated by the `Queue_Stat.java` model (part 1)

Additionally SimJava2 can also produce graphs which can be viewed by the companion tool, the SimJava2 Graph Viewer (SJGV). This will be discussed in more detail in a later lecture note.

SimJava2 also includes the `simdiag` package which is a collection of JavaBeans based classes intended for displaying timing diagrams and graphs of results. However the graphs are based on data gathered from single simulation runs makes them inappropriate for the type of stochastic modelling we are doing in this course, so we will not consider `simdiag`.

12.1.3 Animation

One of the original motivations for the development of SimJava was the provision of easy facilities to develop animations of simulations based on applets. These facilities are provided by the `simanim` package, which is tightly integrated with the `simjava` package. The model is augmented with special trace commands and information about the visual appearance of entities and ports. The traces have two forms, which have differing results on the appearance of the running model.

- `S` traces are used to display the sending of an event by `sim_schedule()`. They have the standard format:

```
"S <port_name> <event_data_string>"
```

In the animation this will display a blue dot (the standard visual representation of a message), with the event data string written next to it, moving along the link from the port named on the scheduling entity.

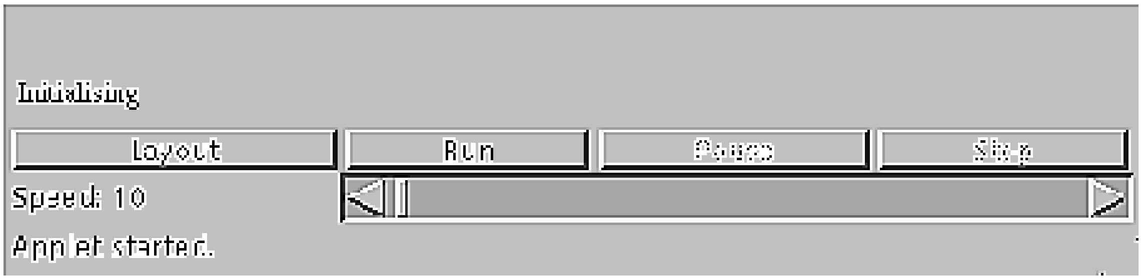
- `P` traces are used to display parameter values associated with entities; these parameters may be dynamically updated as the model evolves. They have the standard format:

  ```
  "P <param1> <param2> ... <paramN>"
  ```

  when the current entity has N parameters. For example the method call `sim_trace(1, "P busy "+3)`, will result in the entities first parameter being updated to value `busy` and the second to value 3. These parameter values may be displayed explicitly in the animation or may result, for example, in a change in the icon used to represent an entity.

`simanim` and `simjava` run concurrently: the trace commands produced by the `simjava` simulation code are fed into `simanim` to produce the animation.

The role of `Sim_system` is now subsumed within a skeleton applet. Two methods of `simanim` create the animation: `anim_init()` is overridden to add buttons, controls and parameter fields, and `anim_layout()` is extended to position the entity icons on the screen and draw ports. To build a simple animated model the modeller gives a `.gif` image and (x,y) coordinates for each entity. Providing additional `.gif` images allows the entity's icon to change according to its internal state, as described above. (see `AppletQ.java` which is a basic animation of the stochastic queue.) `simanim` provides a standard panel for controlling the simulation run:



This allows you to adjust the speed at which the animation runs.

```
############################################################
#              Simulation run measurements                 #
############################################################
----------------------- Customers ------------------------
- Generation rate

Sample mean:          0.10229963671318439
Event count:          11

------------------------ Service -------------------------
- Utilisation

Sample mean:          0.43670097481090003

- Service time

Sample mean:          4.281873568872536
Sample variance:      Not available in efficient mode
Sample std deviation: Not available in efficient mode
Maximum:              8.104271795447396
Minimum:              0.3959786030920327

- Queue length

Sample mean:          0.13959070039740498
Sample variance:      Not available in efficient mode
Sample std deviation: Not available in efficient mode
Maximum:              2.0
Minimum:              0.0
Exceedence proportions:
      0.0 < Queue length <= 1.0 : 0.10242505109839929
      1.0 < Queue length <= 2.0 : 0.018582824649502838
      2.0 < Queue length : 0.0
```

Figure 29: Report generated by the Queue_Stat.java model (part 2)

```
class Source extends Sim_entity {
  private Sim_port enqueue;
  private Sim_negexp_obj src_hold;

  public Source(String name, double iatime) {
    super(name, "source", 15, 100);
    enqueue = new Sim_port("enqueue", "port", Anim_port.RIGHT, 30);
    add_port(enqueue);
    add_param(new Anim_param("Count", Anim_param.VALUE, "0", 10,10));
    src_hold = new Sim_negexp_obj("iatime", iatime);
    add_generator(src_hold);
  }

  public void body() {
    int count;

    while (Sim_system.running()) {
      sim_schedule(enqueue, 0.0, 0);
      sim_trace(1,"S enqueue");
      sim_trace(1,"P ‘‘+(++count));
      sim_pause(src_hold.sample());
    }
  }
}
```

Figure 30: The modified `Source` entity to work with animation.

12.2 Compiling and running SimJava2 models

If you want to run SimJava2 on a home machine you can download all the code from the SimJava2 web page (`http://www.dcs.ed.ac.uk/home/simjava`) and follow the instructions which come with the zip file. Please make sure that you download **SimJava2**, not SimJava1.2.

To compile and run a SimJava2 model under DICE you need to add the `simjava` package to your `CLASSPATH` environment variable. This can be done simply by typing:

```
export CLASSPATH=/usr/java/lib/simjava/classes/:$CLASSPATH
```

To compile a model in a file `Model.java` type:

```
javac Model.java
```

This will produce a class file for each class within the model.
The model can then be executed by typing:

```
java Model
```

When you want to run an animation in the appletviewer you need to add a symbolic link to the animation classes in the directory from which you will be running the appletviewer. This can be done by typing:

```
ln -s /usr/java/lib/simjava/classes/eduni eduni
```

As usual, you are strongly encouraged to develop your understanding by experimenting with the example models provided via the module web page.