Big Data Simulation using ScalaTion

John A. Miller
Department of Computer Science
University of Georgia

March 2, 2014

1 Introduction to Simulation

ScalaTion supports multi-paradigm modeling that can be used for simulation, optimization and analytics. The focus of this document is simulation modeling. Viewed as black-box, a simple *model* maps an input vector $\mathbf{x}$ and a scalar time t to an output/response vector $\mathbf{y}$.

$$\mathbf{y} \;=\; \mathbf{f}(\mathbf{x}, t)$$

A *simulation model* adds to these the notion of state, represented by a vector-valued function $\mathbf{s}(t)$. Knowledge about a system or process is used to define state as well as how state can change over time. Theoretically, this should make such models more accurate, more robust, and have more explanatory power. Ultimately, we may still be interested in how inputs affect outputs, but to increase the realism of the model with the hope of improving its accuracy, much attention must be directed in the modeling effort to state and state transitions. This is true to a degree with most simulation modeling paradigms or world views.

The most recent version of the Discrete-event Modeling Ontology (DeMO) lists five simulation modeling paradigms or world-views for simulation (see the bullet items below). These paradigms are briefly discussed below and explained in detail in [**?**].

- **State-Oriented Models**. State-oriented models, including Generalized Semi-Markov Processes (GSMPs), can be defined using three functions,
  - an activation function $\{e\} = a(\mathbf{s}(t))$,
  - a clock function $t' = c(\mathbf{s}(t), e)$ and
  - a state-transition function $\mathbf{s}(t') = \mathbf{d}(\mathbf{s}(t), e)$.

  In simulation, advancing to the current state $\mathbf{s}(t)$ causes a set of events $\{e\}$ to be activated according to the activation function a. Events occur instantaneously and may affect both the clock and transition functions. The clock function c determines how time advances from t to t' and the state-transition function determines the next state $\mathbf{s}(t')$. In this paper we tie in the input and output vectors. The input vector $\mathbf{x}$ is used to initialize a state at some start time t_0 and the response vector $\mathbf{y}$ can be a function of the state sampled at multiple times during the execution of the simulation model.

- **Event-Oriented Models**. State-oriented models may become unwieldy when the state-space becomes very large. One option is to focus on state changes that occur by processing events in time order. An event may indicate what other events it causes as well as how it may change state. Essentially, the activation and state transition functions are divided into several simpler functions, one for each event e:

- $\{e\} = a_e(\mathbf{s}(t))$ and
- $\mathbf{s}(t') = \mathbf{d_e}(\mathbf{s}(t))$.

Time advance is simplified to just setting the time t' to the time of the most imminent event on a future event list.

- **Process-Oriented Models**. One of the motivations for process-oriented models is that event-oriented models provide a fragmented view of the system or phenomena. As combinations of low-level events determine behavior, it may be difficult to see the big picture or have an intuitive feel for the behavior. Process-oriented or process-interaction models aggregate events by putting them together to form a process. An example of a process is a customer in a store. As the simulated customer (as an active entity) carries out behavior it will conditionally execute multiple events over time. A simulation then consists of many simultaneously active entities and may be implemented using co-routines (or threads/actors as a more heavyweight alternative). One co-routine for each active entity. The overall state of a simulation is then a combination of the states of each active entity and the global shared state, which may include a variety of resources types.

- **Activity-Oriented Models**. There are many types of activity-oriented models including Petri-Nets and Activity-Cycle Diagrams. The main characteristics of such models is a focus on the notion of activity. An activity (e.g, customer checkout) corresponds to a distinct action that occurs over time and includes a start event and an end event. Activities may be started because time advances to its start time or a triggering condition becomes true. Activities typically involve one or more entities. State information is stored in activities, entities and the global shared state.

- **System Dynamics Models**. System dynamics models were recently added to DeMO, since hybrid models that combine continuous and discrete aspects are becoming more popular. In this section, modeling the flight of a golf ball is considered. Let the response vector $\mathbf{y} = [y_0\ y_1]$ where y_0 indicates the horizontal distance traveled, while y_1 indicates the vertical height of the ball. Future positions $\mathbf{y}$ depends on the current position and time t. Using Newton's Second Law of Motion, $\mathbf{y}$ can be estimated by solving a system of Ordinary Differential Equations (ODEs) such as

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, t),\ \mathbf{y}(0) = \mathbf{y_0}.$$

The `Newtons2nd` object uses the Dormand-Prince ODE solver to solve this problem. More accurate models for estimating how far a golf ball will carry when struck by a driver can be developed based on inputs/factors such as club head speed, spin rate, smash factor, launch angle, dimple patterns, ball compression characteristics, etc. There have been numerous studies of this problem, including [**?**].

In addition to these main modeling paradigms, ScalaTion support a simpler approach called Tableau Oriented Models.

2 Tableau Oriented

In tableau oriented simulation models, each simulation entity's event times are recorded in a row of a matrix/tableau. For example in a Bank simulation, each row would store information about a particular customer, e.g., when they arrived, how long they waited, their service time duration, etc. If 10 customers are simulated, the matrix will have 10 rows. Average waiting and service times can be easily calculated by summing columns and dividing by the number of customers. This approach is similar to, but not as flexible as Spreadsheet simulation. The complete code for this example may be found in `Bank`.

```
object Bank extends App
{
    val stream     = 1                                      // random number stream (0 to 99)
    val lambda     = 6.0                                    // customer arrival rate (per hour)
    val mu         = 7.5                                    // customer service rate (per hour)
    val maxCusts   = 10                                     // stopping rule: simulate maxCusts
    val iArrivalRV = Exponential (HOUR/lambda, stream)      // inter-arrival time random variate
    val serviceRV  = Exponential (HOUR/mu, stream)          // service time random variate
    val label      = Array ("ID-0", "IArrival-1", "Arrival-2", "Start-3", "Service-4",
                            "End-5", "Wait-6", "Total-7")
    val mm1 = new Model ("M/M/1 Queue", maxCusts, Array (iArrivalRV, serviceRV), label)
    mm1.simulate ()
    mm1.report

} // Bank
```

2.1 Tableau.scala

The `Model` class support tableau oriented simulation models in which each simulation entity's events are recorded in tabular form (in a matrix). This is analogous to Spreadsheet simulation (http://www.informs-sim.org/wsc06papers/002.pdf).

Class Methods:

```
@param name   the name of simulation model
@param m      the number entities to process before stopping
@param rv     the random variate generators to use
@param label  the column labels for the matrix

class Model (name: String, m: Int, rv: Array [Variate], label: Array [String])

def simulate ()
def report
```

3 Event Oriented

ScalaTion supports two types of event oriented simulation modeling paradigms: Event Scheduling and its extension, called Event Graphs. For both paradigms, the state of the system only changes at discrete event times with the changes specified via event logic. A scheduler within the model will execute the events in time order. A time-ordered priority queue is used to hold the future events and is often referred to as a Future Event List (FEL). Event Graphs capture the event logic related to triggering other events in causal links. In this way, Event Graph models are more declarative (less procedural) than Event Scheduling models. They also facilitate a graphical representation and animation.

3.1 Event Scheduling

A simple, yet practical way to develop a simulation engine to support discrete-event simulation is to implement event-scheduling. This involves creating the following three classes: `Event, Entity` and `Model`. An `Event` is defined as an instantaneous occurrence that can trigger other events and/or change the state of the simulation. An `Entity`, such as a customer in a bank, flows through the simulation. The `Model` serves as a container/controller for the whole simulation and carries out scheduling of event in time order.

For example, to create a simple bank simulation model, one could use the three classes defined in the event-scheduling engine to create subclasses of `Event`, called `Arrival` and `Departure`, and one subclass of `Model`, called `BankModel`. The complete code for this example may be found in `Bank`.

The event logic is coded in the `occur` method which in general triggers future events and updates the current state. It indicates what happens when the event occurs. For the `Arrival` class, the `occur` method will schedule the next arrival event (up to the limit), check to see if the teller is busy. If so, it will place itself in the wait queue, otherwise it schedule its own departure to correspond to its service completion time. Finally, it adjusts the state by incrementing both the number of arrivals (`nArr`) and the number in the system (`nIn`).

```
@param customer  the entity that arrives, in this case a bank customer

case class Arrival (customer: Entity) extends Event (customer, this)      // entity, model
{
    def occur ()
    {
        if (nArr < nArrivals-1) {
            val iArrivalT = iArrivalRV.gen
            val next2Arrive = Entity (clock + iArrivalT, serviceRV.gen)   // next customer
            schedule (iArrivalT, Arrival (next2Arrive))
        } // if
        if (nIn > 0) {                                                    // teller is busy
            waitQueue.enqueue (customer)
        } else {
            t_q_stat.tally (0.0)
            t_s_stat.tally (schedule (customer.serviceT, Departure (customer)))
        } // if
        nArr += 1                                                         // update the current state
        nIn  += 1
    } // occur

} // Arrival class
```

For the `Departure` class, the `occur` method will check to see if there is another customer waiting in the queue and if so, schedule that customer's departure. It will then signal its own departure by updating the state; in this case decrementing `nIn` and incrementing `nOut`.

```
@param customer  the entity that departs, in this case a bank customer

case class Departure (customer: Entity) extends Event (customer, this)    // entity, model
{
    def occur ()
    {
        t_y_stat.tally (clock - customer.arrivalT)
        if (nIn > 1) {
            val next4Service = waitQueue.dequeue ()                 // first customer in queue
            t_q_stat.tally (clock - next4Service.arrivalT)
            t_s_stat.tally (schedule (next4Service.serviceT, Departure (next4Service)))
        } // if
        nIn  -= 1                                                   // update the current state
        nOut += 1
    } // occur

} // Departure class
```

In order to collect statistical information, the `occur` methods of both event classes call the `tally` method from the `Statistics` class to obtain statistics on the **t**ime in **q**ueue `t_q_stat`, the **t**ime in **s**ervice `t_s_stat` and the **t**ime in s**y**stem `t_y_stat`.

The three classes used for creating simulation models following the Event Scheduling paradigm are discussed in the next three subsections.

3.1.1 Event.scala

The `Event` class provides facilities for defining simulation events. A subclass (e.g., `Arrival`) of `Event` must provide event-logic in the implementation of its `occur` method. The `Event` class also provides methods for comparing act times for events and converting an event to its string representation. Note: unique identification and the event/activation time (`actTime`) are mixed in via the `PQItem` trait.

Class Methods:

```
@param entity    the entity involved in this event
@param director  the controller/scheduler that this event is a part of
@param proto     the prototype (serves as node in animation) for this event

abstract class Event (val entity: Entity, director: Model, val proto: Event = null)
         extends PQItem with Ordered [Event]

def compare (ev: Event): Int = ev.actTime compare actTime
def occur ()
override def toString = entity.toString + "\t" + me
```

3.1.2 Entity.scala

An instance of the `Entity` class represents a single simulation entity for event oriented simulation. For each instance, it maintains information about that entity's arrival time and next service time.

Class Methods:

```
@param arrivalT  the time at which the entity arrived
@param serviceT  the amount of time required for the entity's next service

case class Entity (val arrivalT: Double, var serviceT: Double)

override def toString = "Entity-" + eid
```

3.1.3 Model.scala

The `Model` class schedules events and implements the time advance mechanism for event oriented simulation models. It provides methods to `schedule` and `cancel` events. Scheduled events are place in the Future Event List (FEL) in time order. The `simulate` method will cause the main simulation loop to execute, which will remove the most imminent event from the FEL and invoke its `occur` method. The simulation will continue until a stopping rule evaluates to true. Methods to `getStatistics` and `report` statistical results are also provided.

Class Methods:

```
@param name       the name of the model
@param animation  whether to animate the model (only for Event Graphs)

class Model (name: String, animation: Boolean = false)
      extends ModelT with Identity

def schedule (timeDelay: Double, event: Event): Double =
def cancel (event: Event)
def simulate (startTime: Double = 0.0): ListBuffer [Statistic] =
def report (eventType: String, links: Array [CausalLink] = Array ())
def report (vars: Array [Tuple2 [String, Double]])
def reports (stats: Array [Tuple2 [String, Statistic]])
def getStatistics: ListBuffer [Statistic] =
def animate (who: Identity, what: Value, color: Color, shape: Shape, at: Array [Double])
def animate (who: Identity, what: Value, color: Color,
             shape: Shape, from: Event, to: Event, at: Array [Double] = Array ())
```

The `animate` methods are used with Event Graphs (see the next section).

3.2 Event Graphs

Event Graphs operate in a fashion similar to Event Scheduling. Originally proposed as a graphical conceptual modeling technique (Schruben, 1983) for designing event oriented simulation models, modern programming languages now permit more direct support for this style of simulation modeling.

In ScalaTion, the simulation engine for Event Graphs consists of the following four classes: `Entity`, `Model`, `EventNode` and `CausalLink`. The first two are shared with Event Scheduling. An `Entity`, such as a customer in a bank, flows through the simulation. The `Model` serves as a container/controller for the whole simulation. The last two are specify to Event Graphs. An `EventNode` (subclass of `Event`), defined as an instantaneous occurrence that can trigger other events and/or change the state of the simulation, is represented as a *node* in the event graph. A `CausalLink` emanating from an event/node is represented as an outgoing directed *edge* in the event graph. It represents causality between events. One event can conditionally trigger another event to occur some time in the future.

For example, to create a simple bank simulation, one could use the four classes provided by the Event Graph simulation engine to create subclasses of `EventNode`, called `Arrival` and `Departure`, and one subclass of `Model`, called `BankModel`. The complete code for this example may be found in `Bank2`. In more complex situations, one would typically define a subclass of `Entity` to represent the customers in the bank.

```
class BankModel (name: String, nArrivals: Int, arrivalRV: Variate, serviceRV: Variate)
      extends Model (name)
```

The Scala code below was made more declarative than typical code for event-scheduling to better mirror event graph specifications, where the causal links specify the conditions and time delays. For instance,

```
() => nArr < nArrivals
```

is a closure returning `Boolean` that will be executed when arrival events are handled. In this case, it represents a stopping rule; when the number of arrivals exceeds a threshold, the arrival event will no longer schedule the next arrival. The `serviceRV` is a random variate to be used for computing service times.

In the `BankModel` class, one first defines the state variables: `nArr, nIn` and `nOut`. For animation of the event graph, a prototype for each type of event is created and displayed as a node. The edges connecting these prototypes represent the casual links. The `aLinks` array holds two causal links emanating from `Arrival`, the first a self link representing triggered arrivals and the second representing an arrival finding an idle server, so it can schedule its own departure. The `dLinks` array holds one causal link emanating from `Departure`, a self link representing the departing customer causing the next customer in the waiting queue to enter service (i.e., have its departure scheduled).

```
//:: define the state variables for the simulation

var nArr = 0.0                          // number of customers that have arrived
var nIn  = 0.0                          // number of customers in the bank
var nOut = 0.0                          // number of customers that have finished and left the bank

//:: define the nodes in the event graph (event prototypes)

val protoArrival   = Arrival (null)     // prototype for all Arrival events
val protoDeparture = Departure (null)   // prototype for all Departure events

//:: define the edges in the event graph (causal links between events)
```

```
val aLinks = Array (CausalLink ("link2A", this, () => nArr < nArrivals, protoArrival,
                                () => Arrival (null), arrivalRV),
                    CausalLink ("link2D", this, () => nIn == 0, protoDeparture,
                                () => Departure (null), serviceRV))
val dLinks = Array (CausalLink ("link2D", this, () => nIn > 1, protoDeparture,
                                () => Departure (null), serviceRV))

protoArrival.displayLinks (aLinks)
protoDeparture.displayLinks (dLinks)
```

An animation of the Event Graph consisting of two `EventNodes` `Arrival` and `Departure` and three `CausalLinks` is depicted in Figure 1.

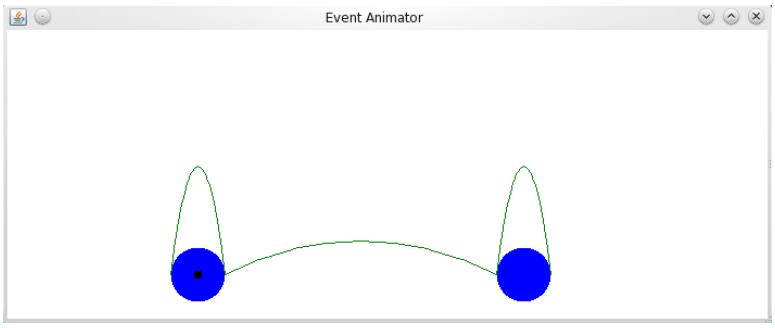

Figure 1: Event Graph Animation of a Bank.

The main thing to write within each subclass of `EventNode` is the `occur` method. To handle arrival events, the `occur` method of the `Arrival` class first calls the `super.occur` method from the superclass to trigger other events using the causal links and then updates the state by incrementing both the number of arrivals (`nArr`) and the number in the system (`nIn`).

```
@param customer  the entity that arrives, in this case a customer

case class Arrival (customer: Entity)
     extends EventNode (customer, this, protoArrival, Array (150.0, 200.0, 50.0, 50.0), aLinks)
{
    override def occur ()
    {
        super.occur ()   // handle casual links
        nArr += 1        // update the current state
        nIn  += 1
    } // occur

} // Arrival class
```

To handle departure events, the `occur` method `Departure` class first calls the `occur` method of the superclass to trigger other events using the causal links and then updates the state by decrementing the number in the system (`nIn`) and incrementing the number of departures (`nOut`).

```
@param customer  the entity that departs, in this case a customer

case class Departure (customer: Entity)
     extends EventNode (customer, this, protoDeparture, Array (450.0, 200.0, 50.0, 50.0), dLinks)
{
    override def occur ()
    {
        super.occur ()   // handle casual links
        nIn  -= 1        // update the current state
        nOut += 1
    } // occur

} // Departure class
```

Two of the three classes used for creating simulation models following the Event Scheduling paradigm can be used for Event Graphs, namely `Entity` and `Model`. `Event` must be replaced with its subclass called `EventNode`. These form the nodes in the Event Graphs. An edge in the Event Graph is an instance of the `CausalLink` class. These two new classes (`EventNode` and `CausalLink`) are described in the subsections below.

3.2.1 EventNode.scala

The 'Event' class provides facilities for defining simulation events. Subclasses of Event provide event-logic in their implementation of the occur method. Note: unique identification and the event/activation time (actTime) are mixed in via the PQItem trait.

Class Methods:

```
@param proto     the prototype (serves as node in animation) for this event
@param entity    the entity involved in this event
@param links     the causal links used to trigger other immediate/future events
@param director  the controller/scheduler that this event is a part of
@param at        the location of this event

abstract class EventNode (val proto: Event, entity: Entity, links: Array [CausalLink],
                          director: Model, at: Array [Double] = Array ())
         extends PQItem with Ordered [Event]

def compare (ev: Event): Int = ev.actTime.compare (actTime)
def occur ()
def display ()
def displayLinks (outLinks: Array [CausalLink])
```

3.2.2 CausalLink.scala

The 'CausalLink' class provides casual links between events. After an event has updated the state, it checks its causal links to schedule/cancel other events.

Class Methods:

```
@param _name      the name of the causal link
```

```
@param condition  the condition under which it is followed
@param makeEvent  function to create an event
@param delay      the time delay in scheduling the event
@param cancel     whether to schedule (default) or cancel the event

case class CausalLink (_name: String, director: Model, condition: () => Boolean, causedEvent: Event,
                       makeEvent: () => Event, delay: Variate, cancel: Boolean = false)
      extends Identity
def display (from: Event, to: Event)
def tally (duration: Double) { _durationStat.tally (duration) }
def accumulate (value: Double, time: Double) { _persistentStat.accumulate (value, time) }
def durationStat = _durationStat
def persistentStat = _persistentStat
```

4 Process Interaction

Many discrete-event simulation models are written using the process-interaction world view, because the code tends to be concise and intuitively easy to understand. Take for example the process-interaction model of a bank (`BankModel` a subclass of `Model`) shown below. Following this world view, one simply constructs the simulation components and then provides a script for entities (`SimActors`) to follow while in the system. In this case, the `act` method for the customer class provides the script (what entities should do), i.e., enter the bank, if the tellers are busy wait in the queue, then receive service and finally leave the bank.

The development of a simulation engine for process-interaction models is complicated by the fact that concurrent (or at least quasi-concurrent) programming is required. Various language features/capabilities from lightweight to middleweight include continuations, coroutines, actors and threads. Heavyweight concurrency via OS processes is infeasible, since simulations may require a very large number of concurrent entities. The main requirement is for a concurrent entity to be able to suspend its execution and be resumed where it left off (its state being maintained on a stack). Since preemption is not necessary, lightweight concurrency constructs are ideal. Presently, ScalaTion uses Scala Actors for concurrency. Future implementations will include use of continuations and Akka Actors.

ScalaTion includes several types of model components: `Gate, Junction, Resource, Route, Sink, Source, Transport` and `WaitQueue`. A model may be viewed as a directed graph with several types of nodes:

- `Gate`: a gate is used to control the flow of entities, they cannot pass when it is shut.
- `Junction`: a junction is used to connect two transports.
- `Resource`: a resource provides services to entities (typically resulting in some delay).
- `Sink`: a sink consumes entities.
- `Source`: a source produces entities.
- `WaitQueue`: a wait-queue provides a place for entities to wait, e.g., waiting for a resource to become available or a gate to open.

These nodes are linked together with directed edges (from, to) that model the flow entities from node to node. A `Source` node must have no incoming edges, while a `Sink` node must have no outgoing edges.

- `Route`: a route bundles multiple transports together (e.g., a two-lane, one-way street).
- `Transport`: a transport is used to move entities from one component node to the next.

The model graph includes coordinates for the component nodes to facilitate animation of the model. Coordinates for the component edges are calculated based on the coordinates of its from and to nodes. Small colored tokens move along edges and jump through nodes as the entities they represent flow through the system.

The `BankModel` may be developed as follows: The `BankModel` first defines the component nodes `entry, tellerQ, teller,` and `door`. Then two edge components, `toTellerQ` and `toDoor`, are defined. These six components are added to the `BankModel` using the `addComponent` method. Note, the endpoint nodes for an edge must be added before the edge itself. Finally, a inner case class called `Customer` is defined where the `act` method specifies the script for bank customers to follow. The `act` method specifies the behavior of concurrent entities (Scala Actors) and is analogous to the `run` method for Java/Scala Threads.

```
class BankModel (name: String, nArrivals: Int, iArrivalRV: Variate,
                 nUnits: Int, serviceRV: Variate, moveRV: Variate)
      extends Model (name)
{
    val entry     = Source ("entry", this, Customer, 0, nArrivals, iArrivalRV, (100, 290))
    val tellerQ   = WaitQueue ("tellerQ", (330, 290))
    val teller    = Resource ("teller", tellerQ, nUnits, serviceRV, (350, 285))
    val door      = Sink ("door", (600, 290))
    val toTellerQ = new Transport ("toTellerQ", entry, tellerQ, moveRV)
    val toDoor    = new Transport ("toDoor", teller, door, moveRV)

    addComponent (entry, tellerQ, teller, door, toTellerQ, toDoor)

    case class Customer () extends SimActor ("c", this)
    {
        def act ()
        {
            toTellerQ.move ()
            if (teller.busy) tellerQ.waitIn () else tellerQ.noWait ()
            teller.utilize ()
            teller.release ()
            toDoor.move ()
            door.leave ()
        } // act

    } // Customer

} // BankModel class
```

Note, that the bank model for event-scheduling did not include time delays and events for moving token along transports. In `BankModel2`, the impact of transports is reduced by (1) using the transport's `jump` method rather than its `move` method and (2) reducing the time through the transport by an order of magnitude. The `jump` method has the tokens jumping directly to the middle of the transport, while the `move` method simulates smooth motion using many small hops. Both `BankModel` and `BankModel2` are in the `apps.process` package as well as `CallCenterModel`, `ERoomModel`, `IntersectionModel`, `LoopModel` `MachineModel` and `RoadModel`.

4.1 Component.scala

The `Component` trait provides basic common feature for simulation components. A component may function either as a node or edge. Entities/sim-actors interact with component nodes and move/jump along component edges. All components maintain sample/duration statistics (e.g., time in waiting queue) and all except `Gate`, `Source` and `Sink` maintain time-persistent statistics (e,g., number in waiting queue).

Class Methods:

```
trait Component extends Identity

def initComponent (label: String, loc: Array [Double])
def initStats (label: String)
def director = _director
def setDirector (dir: Model)
```

```
def display ()
def tally (duration: Double) { _durationStat.tally (duration) }
def accumulate (value: Double, time: Double) { _persistentStat.accumulate (value, time) }
def durationStat = _durationStat
def persistentStat = _persistentStat
```

4.2 Signifiable.scala

The `Signifiable` trait defines standard messages sent between actors implementing process interaction simulations.

Class Methods:

```
trait Signifiable
```

4.3 SimActor.scala

The `SimActor` abstract class represents entities that are active in the model. The `act` abstract method, which specifies entity behavior, must be defined for each subclass. Each `SimActor` extends Scala's `Actor` class and may be roughly thought of as running in its own thread. The script for entities/sim-actors to follow is specified in the `act` method of the subclass as was done for the `Customer` case class in the `BankModel`.

Class Methods:

```
@param name      the name of the entity/SimActor
@param director  the director controlling the model

abstract class SimActor (name: String, director: Model)
         extends Actor with Signifiable with PQItem with Ordered [SimActor] with Locatable

def subtype: Int = _subtype
def setSubtype (subtype: Int) { _subtype = subtype }
def trajectory: Double = traj
def setTrajectory (t: Double) { traj = t }
def compare (actor2: SimActor): Int = actor2.actTime compare actTime
def act ()
def yetToAct = _yetToAct
def nowActing () { _yetToAct = false }
def time = director.clock
def schedule (delay: Double)
def yieldToDirector (quit: Boolean = false)
```

4.4 Source.scala

The `Source` class is used to periodically inject entities (`SimActors`) into a running simulation model (and a token into the animation). It may act as an arrival generator. A `Source` is both a simulation `Component` and a special `SimActor`, and therefore can run concurrently.

Class Methods:

```
@param name          the name of the source
@param director      the director controlling the model
@param makeEntity    the function to make entities of a specified type
@param subtype       indicator of the subtype of the entities to me made
@param units         the number of entities to make
@param iArrivalTime  the inter-arrival time distribution
@param at            the location of the source (x, y, w, h)

class Source (name: String, director: Model, makeEntity: () => SimActor, subtype: Int, units: Int,
              iArrivalTime: Variate, at: Array [Double])
      extends SimActor (name, director) with Component

def this (name: String, director: Model, makeEntity: () => SimActor, units: Int,
def display ()
def act ()
```

4.5 Sink.scala

The `Sink` class is used to terminate entities (`SimActors`) when they are finished. This class will remove the token from the animation and collect important statistics about the entity.

Class Methods:

```
@param name  the name of the sink
@param at    the location of the sink (x, y, w, h)
class Sink (name: String, at: Array [Double])
      extends Component

def this (name: String, director: Model, makeEntity: () => SimActor, subtype: Int, units: Int,
          iArrivalTime: Variate, xy: Tuple2 [Double, Double])
def display ()
def leave ()
```

4.6 Transport.scala

The `Transport` class provides a pathway between two other component nodes. The `Component`s in a `Model` conceptually form a graph in which the edges are `Transport` objects and the nodes are other `Component` objects. An edge may be either a `Transport` or `Route`.

Class Methods:

```
@param name      the name of the transport
@param from      the first/starting component
@param to        the second/ending component
@param motion    the speed/trip-time to move down the transport
@param isSpeed   whether speed or trip-time is used for motion
@param bend      the bend or curvature of the transport (0 => line)
@param shift1    the x-y shift for the transport's first endpoint (from-side)
@param shift2    the x-y shift for the transport's second endpoint (to-side)

class Transport (name: String, val from: Component, val to: Component,
```

```
                motion: Variate, isSpeed: Boolean = false,
                bend: Double = 0.0, shift1: R2 = R2 (0.0, 0.0), shift2: R2 = R2 (0.0, 0.0))
      extends Component

def display ()
override def at: Array [Double] =
def jump ()
def move ()
```

4.7 Resource.scala

The `Resource` class provides services to entities (`SimActors`). The service provided by a resource typically delays the entity by an amount of time corresponding to its service time. The `Resource` may or may not have an associated waiting queue.

Class Methods:

```
@param name         the name of the resource
@param line         the line/queue where entities wait
@param units        the number of service units (e.g., bank tellers)
@param serviceTime  the service time distribution
@param at           the location of the resource (x, y, w, h)

class Resource (name: String, line: WaitQueue, private var units: Int, serviceTime: Variate,
                at: Array [Double])
      extends Component

def this (name: String, line: WaitQueue, units: Int, serviceTime: Variate,
          xy: Tuple2 [Double, Double])
def changeUnits (dUnits: Int)
def display ()
def busy = inUse == units
def utilize ()
def utilize (duration: Double)
def release ()
```

4.8 WaitQueue.scala

The `WaitQueue` class is a wrapper for Scala's Queue class, which supports FCSC Queues. It adds monitoring capabilities and optional capacity restrictions. If the queue is full, entities (`SimActors`) attempting to enter the queue are barred. At the model level, such entities may be (1) held in place, (2) take an alternate route, or (3) be lost (e.g., dropped call/packet). An entity on a `WaitQueue` is suspended for an indefinite wait. The actions of some other concurrent entity will cause the suspended entity to be resumed (e.g., when a bank customer finishes service and releases a teller).

Class Methods:

```
@param name  the name of the wait-queue
@param at    the location of the wait-queue (x, y, w, h)
@param cap   the capacity of the queue (defaults to unbounded)
```

```
class WaitQueue (name: String, at: Array [Double], cap: Int = Int.MaxValue)
      extends Queue [SimActor] with Component

def this (name: String, xy: Tuple2 [Double, Double], cap: Int)
def isFull: Boolean = length >= cap
def barred: Int = _barred
def display ()
def waitIn ()
def noWait ()
```

4.9 Junction.scala

The `Junction` class provides a connector between two transports/routes. Since `Line`s and `QCurve`s have limitation (e.g., hard to make a loop back), a junction may be needed.

Class Methods:

```
@param name  the name of the junction
@param director  the director controlling the model
@param jTime     the jump-time through the junction
@param at        the location of the junction (x, y, w, h)

class Junction (name: String, director: Model, jTime: Variate, at: Array [Double])
      extends Component

def this (name: String, director: Model, jTime: Variate, xy: Tuple2 [Double, Double])
def display ()
def move ()
```

4.10 Gate.scala

The `Gate` class models the operation of gates that can open and shut. When a gate is open, entities can flow through and when shut, they cannot. When shut, the entities may wait in a queue or go elsewhere. A gate can model a traffic light (green $\Longrightarrow$ open, red $\Longrightarrow$ shut).

Class Methods:

```
@param name      the name of the gate
@param director  the model/container for this gate
@pram  line      the queue holding entities waiting for this gate to open
@param units     number of units/phases of operation
@param onTime    distribution of time that gate will be open
@param offTime   distribution of time that gate will be closed
@param at        the location of the Gate (x, y, w, h)
@param shut0     Boolean indicating if the gate is opened or closed
@param cap       the maximum number of entities that will be released when the gate is opened

class Gate (name: String, director: Model, line: WaitQueue, units: Int, onTime: Variate, offTime: Variate,
            at: Array [Double], shut0: Boolean, cap: Int = 10)
      extends SimActor (name, director) with Component
```

```
def this (name: String, director: Model, line: WaitQueue, units: Int, onTime: Variate, offTime: Variate,
          xy: Tuple2 [Double, Double], shut0: Boolean, cap: Int)
def shut: Boolean = _shut
def display ()
def release ()
def act ()
def gateColor: Color = if (_shut) red else green
def flip () { _shut = ! _shut }
def duration: Double = if (_shut) offTime.gen else onTime.gen
```

4.11 Route.scala

The `Route` class provides a multi-lane pathway between two other node components. The `Component`s in a `Model` conceptually form a graph in which the edges are `Transport`s/`Route`s and the nodes are other components. A route is a composite component that bundles several transports.

Class Methods:

```
@param name      the name of the route
@param k         the number of lanes/transports in the route
@param from      the starting component
@param to        the ending component
@param motion    the speed/trip-time to move down the transports in the route
@param isSpeed   whether speed or trip-time is used for motion
@param angle     angle in radians of direction (0 => east, Pi/2 => north, Pi => west, 3Pi/2 => south)
@param bend      the bend or curvature of the route (0 => line)

class Route (name: String, k: Int, from: Component, to: Component,
             motion: Variate, isSpeed: Boolean = false,
             angle: Double = 0.0, bend: Double = 0.0)
      extends Component

override def at: Array [Double] = lane(0).at
def display ()
```

4.12 Model.scala

The `Model` class maintains a list of components making up the model and controls the flow of entities (`SimActors`) through the model, following the process-interaction world-view. It maintains a time-ordered priority queue to activate/re-activate each of the entities. Each entity (`SimActor`) is implemented as a Scala `Actor` and may be roughly thought of as running in its own thread.

Class Methods:

```
@param name       the name of the model
@param animating  whether to animate the model

class Model (name: String, animating: Boolean = true)
      extends Actor with Signifiable with Modelable with Component

def addComponent (_parts: Component*) { for (p <- _parts) parts += p }
```

```
def addComponents (_parts: List [Component]*) { for (p <- _parts; q <- p) parts += q }
def theActor = _theActor
def simulate (startTime: Double = 0.0)
def reschedule (actor: SimActor) { agenda += actor }
def act ()
def report
def reportf { new StatTable (name + " statistics", getStatistics) }
def getStatistics: ListBuffer [Statistic] =
def display ()
def animate (who: Identifiable, what: Value, color: Color, shape: Shape, at: Array [Double])
def animate (who: Identifiable, what: Value, color: Color, shape: Shape,
             from: Component, to: Component, at: Array [Double] = Array ())
```