Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

AN INTRODUCTORY TUTORIAL FOR THE KOTLIN SIMULATION LIBRARY

Manuel D. Rossetti[1]

[1]Department of Industrial Engineering, University of Arkansas, Fayetteville, AR, USA

ABSTRACT

The Kotlin Simulation Library (KSL) is an open-source library written in the Kotlin programming language that facilitates Monte Carlo and discrete-event simulation modeling. This paper provides a tutorial of the functionality of the discrete-event modeling capabilities provided by the KSL. The library provides an API framework for developing, executing, and analyzing models using both the event view and the process view modeling perspectives. Because models can be developed that contain both modeling perspectives, the KSL provides great flexibility during the modeling building process. This tutorial provides both an overview of the library and also a complete example including its analysis using KSL constructs.

1 INTRODUCTION

A full description of the capabilities of the KSL are provided in Rossetti (2023a) with a summary overview provided in Rossetti (2023d). The KSL provides application programming interfaces and classes that extend the capabilities of the Kotlin programming language to include generating pseudo-random numbers, random variate generation, statistical analysis, and discrete-event system simulation model building via both the event view and the process view. The KSL *modeling* and *simulation* packages provide sets of classes and interfaces that facilitate the building of discrete-event simulation models. The *simulation* package contains the *Model* and *ModelElement* classes that form the building blocks of KSL models. The *modeling* package contains specialized model elements (sub-classes of *ModelElement*) that facilitate the modeling of common system components (e.g., resources, queues, entities, etc.) that are typically utilized in simulation modeling situations. The purpose of this paper is to illustrate the usage of these constructs and provide a tutorial to allow persons new to simulation modeling or persons familiar with other simulation languages to build and utilize KSL models.

According to a number of programming language rankings, Kotlin, has been within the top 15 languages for the past 5 years and according to Loftus (2023), Kotlin is currently ranked number 11 and has been steadily increasing in popularity. Kotlin offers a number of interesting language capabilities such as clear and concise functional and object-oriented specifications, static compilation, type-safety, explicit null representation, and asynchronous programming through co-routines that make it preferable to languages such as Python and Java. Building on a long history of leveraging co-routines found in early simulation languages such as Simula and Simscript, the KSL brings the process view to the Java Virtual Machine ecosystem.

The paper is organized as follows. The next section presents background on the overall KSL modeling functionality and capabilities. The purpose of the section is to provide a foundation of terms, constructs, and capabilities that will be illustrated in subsequent sections. Then, section 3 of the paper will illustrate the KSL modeling on a semi-realistic problem situation suitable for a tutorial. Section 3 focuses on the model building process. Then, section 4 will illustrate how to exercise the model and apply KSL functionality for analyzing simulation runs. Finally, the last section provides an overview of additional KSL functionality not discussed in this paper and describes future functionality to enhance the library.

2 OVERVIEW OF KSL MODELING FUNCTIONALITY

The purpose of the KSL is to support education and research within simulation. The KSL's current version has packages that support random number generation, statistical collection, basic reporting, and discrete-event simulation modeling via both the event view and the process view. The KSL's functionality is provided by a number of API packages, including:

- *calendar* - The calendar package implements classes that provide event calendar processing.
- *simulation* - The simulation package implements the main classes involved in constructing and running simulation models.
- *modeling* - The modeling package is the heart of the KSL. In the modeling package, supporting model elements such as queues, resources, variables, responses, etc. are implemented. This package will be illustrated in this paper.
- *observers* - The observers package provides support classes for observing model elements during the simulation run. The KSL is designed to allow each model element to have observers attached to it thereby implementing the classic Observer pattern. Observers can be used to collect statistics, write output to files, display model element state, etc.
- *utilities* - The utilities package provides support classes that are used by the KSL. The random, statistics, reporting, and database related sub-packages are briefly discussed in this paper.

The main focus of this overview is a discussion of the *simulation* and *modeling* packages, which provide the constructs for modeling and executing discrete-event simulation models.

A discrete event system is a system in which the state of the system changes only at discrete points in time. There are essentially two fundamental viewpoints for modeling a discrete event system: the event view and the process view. These views are different representations for the same system. In the event view, the system and its elements are conceptualized as reacting to events. In the process view, the movement of entities through their processes implies the events within the system.

The following classes within the *simulation* package work together to provide for the scheduling and execution of events:

- *Model* - An instance of *Model* holds the model and facilitates the running of the simulation according to run parameters such as simulation run length and number of replications. An instance of a *Model* is required to serve as the parent to any model elements within a simulation model. It is the top-level container for model elements.
- *Executive* - The *Executive* controls the execution of the events and works with the calendar package to ensure that events are executed, and the appropriate model logic is called at the appropriate event time. This class is responsible for placing the events on a calendar, allowing events to be canceled, and executing the events in the correct order.
- *KSLEvent* - This class represents a simulation event. The attributes of *KSLEvent* provide information about the name, time, priority, and type of the event. The user can also check whether or not the event is canceled or if it has been scheduled. In addition, a generic attribute of type `<T>` can be associated with an event and can be used to pass information along with the event.
- *ModelElement* - This class serves as a base class for all classes that are within a simulation model. It provides access to the scheduler (executive) and ensures that model elements participate in common simulation actions (e.g. warm up, initialization, etc.).

To build an event view KSL model, the user creates a sub-class of *ModelElement* that implements the actions associated with events that represent changes in state for the object being modeled. The following code illustrates how to implement a model that represents the arrival of events within a simple Poisson process.

```
class SimplePoissonProcess (parent: ModelElement, name: String? = null) :
    ModelElement(parent, name) {
    private val myTBE: RandomVariable = RandomVariable(this, ExponentialRV(1.0))
    private val myCount: Counter = Counter(this, name = "Counts events")
    private val myEventHandler: EventHandler = EventHandler()

    override fun initialize() {
        super.initialize()
        schedule(myEventHandler, myTBE.value)
    }

    private inner class EventHandler : EventAction<Nothing>() {
        override fun action(event: KSLEvent<Nothing>) {
            myCount.increment()
            schedule(myEventHandler, myTBE.value)
        }
    }
}
```

In this code, we see a class *SimplePoissonProcess* that sub-classes from *ModelElement*. In the first line of the class, a random variable is defined to represent the time between arrivals for the Poisson process. Then, an instance of a *Counter* is declared to count the number of events processed. To implement the event routine, the basic approach is to define an inner class that implements an *EventAction* interface and provide the required *action()* method to implement the change of state within its body.

In the aforementioned code, the inner class *EventHandler* provides the change in state functionality for the Poisson process. In the action method, the counter is incremented, and the event is scheduled according to the time between events. The private attribute *myEventHandler* was declared to reference the event routine. In essence, the reference to the event handler is placed on the calendar when the event is scheduled. Then, the executive ensures that the event's action method is invoked at the appropriate time.

In the *initialize*() method, the logic is provided to schedule the first event. The *initialize()* method is an inherited method from the base *ModelElement* class. The *initialize()* method is called automatically by the simulation executive at the beginning of each replication. Once the event occurs at the scheduled time, the count will be incremented, and the next event scheduled. To execute this simulation model, we need to construct an instance of a model and specify the simulation run parameters. This is illustrated in the following code.

```
fun main() {
    val s = Model("Simple PP")
    SimplePoissonProcess(s.model)
    s.lengthOfReplication = 20.0
    s.numberOfReplications = 50
    s.simulate()
    s.print()
}
```

In this brief code snippet, the *main* function of the program is used to create the model and instance of the Poisson process. The length of the replication and number of replications is specified, and the model is simulated. Finally, the model results are printed to the console. In general, KSL modeling has this general form. First, create classes that sub-class from *ModelElement* and provide the model logic. Then, create an

instance of the *Model* class, attach instances of the sub-classes to the model instance, specify the characteristics of the simulation execution parameters, simulate the model, and output the results. The complexity of the event specification and the subsequent logic needed to implement state changes is the main challenge in modeling using the event view.

The characteristics of a process view simulation model within the KSL follows the same general pattern: create a sub-class that implements the model logic, specify the characteristics of the simulation execution parameters, simulate the model, and output the results. However, instead of sub-classing from *ModelElement*, to implement a process view simulation, the user sub-classes from a specialized sub-class of *ModelElement* called *ProcessModel*. The *ProcessModel* class facilitates the modeling of entities experiencing processes. A *ProcessModel* has inner classes (*Entity*, *EntityGenerator*, etc.) that can be used to describe entities and the processes that they experience. To illustrate the process view, we will discuss the implementation of a simple M/M/1 queuing system (1 server, with Poisson arrivals and exponential service times). Before proceeding, we need to present some additional commonly used KSL constructs that were glossed over in the preceding example.

To make useful simulation models, we need to generate random variables and collect statistics. The KSL has extensive functionality for the generation of random quantities and for collecting statistics both within the Monte Carlo context and within the DEDS context. Rossetti (2023d) provides an overview of these constructs, especially within the Monte Carlo context. In this paper, we overview the most important aspects for use within a DEDS context.

A *RandomVariable* is a sub-class of *ModelElement* that is used to represent random variables within a simulation model. The *RandomVariable* class must be supplied an instance of a class that implements the *RandomIfc* interface. Implementation of the *RandomIfc* interface provides a *value* property that returns a random value and permits random number stream control. Stream control is important when utilizing advanced simulation statistical methods such as common random numbers. For example, stream control is used to advance the state of the underlying stream to the next sub-stream at the end of every replication of the model. This helps in synchronizing the use of random numbers in certain types of experimental setups.

In the previously illustrated code, the time between arrivals was modeled using an instance of the *RandomVariable* class that was supplied an *ExponentialRV* instance. The *RandomVariable* class is a sub-class of *ModelElement*, which hooks instances of the *RandomIfc* interface into a KSL model and automatically controls the streams.

While the *utilities* package provides a plethora of statistical computing functionality, this functionality needs to be accessible within a KSL model. The main statistical collection functionality is provided by the following classes:

- *Counter* - A *Counter* is also a sub-class of *ModelElement* which facilitates the incrementing and decrementing of a variable and the statistical collection of the variable across replications. The value of the variable associated with the instance of a *Counter* is automatically reset to 0.0 at the beginning of each replication.
- *Response* – A *Response* is a sub-class of *ModelElement* that can take on values within a simulation model and allows easy statistical observation of its values during the simulation run. This class provides observation-based statistical collection, which is sometimes called tally statistics in many simulation languages.
- *TWResponse* – A *TWResponse* is a sub-class of *ModelElement* that facilitates the observation and statistical collection of time-persistent variables. *TWResponse* automates the collection of time averages for simulation state variables.

Counters and response variables are used to instrument the model for statistical output. The KSL will automatically tabulate within replication statistics and across replication statistics for these constructs. The KSL provides for detailed reporting of the statistics in multiple formats (text, markdown, csv, etc.) and also

will automatically store the statistical results within a well-designed database for post processing. The use of these constructs will be noted in the following implementation of the M/M/1 queuing situation.

To implement the process view, we must sub-class from *ProcessModel*, specify the entity whose process is being simulated, and provide a description of the process. In the following code, the class *SimpleServiceSystem* sub-classes from the *ProcessModel* class. The input parameters to the class constructor specify the number of servers, the time between arrival random variable, and the service time random variable. The private variables *serviceTime* and *timeBetweenArrivals* are random variables that hook the provided inputs into the model. The variables (*wip, timeInSystem, numCustomers*) are instances of the previously described time-weighted, response, and counter constructs for collecting statistics during the simulation. Lastly, the variable, *servers*, is a new modeling construct for representing resources within KSL process models. A resource is a set of similar units that can be used by entities during a process. In this case, the resource will automatically have an instance of the *Queue* class created to hold entities that are required to wait for the resource when it is not immediately available.

```
class SimpleServiceSystem(
    parent: ModelElement,
    numServers: Int = 1,
    ad: RandomIfc = ExponentialRV(1.0, 1),
    sd: RandomIfc = ExponentialRV(0.5, 2),
    name: String? = null
) : ProcessModel(parent, name) {
    init {
        require(numServers > 0) { "The number of servers must be >= 1" }
    }

    private var serviceTime: RandomVariable = RandomVariable(this, sd)
    private var timeBetweenArrivals: RandomVariable = RandomVariable(parent, ad)
    private val wip: TWResponse = TWResponse(this, "${this.name}:NumInSystem")
    private val timeInSystem: Response = Response(this,
"${this.name}:TimeInSystem")
    private val numCustomers: Counter = Counter(this, "${this.name}:NumServed")
    private val servers: ResourceWithQ = ResourceWithQ(this, "Servers", numServers)
```

Similar to the example of the Poisson process, we need to implement the arrival of the customers to the system. To keep the presentation simple, the implementation is similar to the previous example. In the following code, the *arrival()* function acts like the previously presented event handler for the arrival events. This form takes advantage of the functional programming capabilities of Kotlin. The *initialize()* method schedules the first arrival event. When the arrival event executes, it creates a customer, activates the customer's process, and schedules the next arrival. The notation (*this::arrival*) is a function reference.

```
override fun initialize() {
    schedule(this::arrival, timeBetweenArrivals)
}

private fun arrival(event: KSLEvent<Nothing>) {
    val c = Customer()
    activate(c.pharmacyProcess)
    schedule(this::arrival, timeBetweenArrivals)
}
```

```
private inner class Customer : Entity() {
    val serviceProcess: KSLProcess = process() {
        wip.increment()
        timeStamp = time
        val a = seize(servers)
        delay(serviceTime)
        release(a)
        timeInSystem.value = time - timeStamp
        wip.decrement()
        numCustomers.increment()
    }
}
```

The process view is encapsulated within the inner class *Customer*, which sub-classes from *Entity*, and defines the customer's process. Let's go through the form of this process description. In the first two lines of the service process, the work-in-process (wip) is incremented and the time of the arrival of the customer is noted with a time stamp. Then, the use of the resource is described with the (*seize*, *delay*, and *release*) suspending functions. The variable "a" returned from the *seize()* function is the allocation of the named resource to the entity. This allocation is then used in the *release()* function, after the entity has completed the delay for the service.

The *process()* function is the mechanism used to describe a co-routine. The co-routine permits the use of suspending functions, which cause execution to suspend at that point to later return. This is the essence of process view modeling. It is important to understand that simulated time is passing while the process is suspended. After the resource is used, the time spent in the system is collected, the work-in-process decremented, and the total number of customers processed in incremented. As one might note, the KSL process modeling functionality has a similar form as that found in some commercial simulation packages.

To setup and run this model, the following code can be used. Notice that the number of replications, length of the replication, and a warmup period can be easily specified. The simple service system instance is instantiated and connected to the model instance, simulated, and results printed.

```
fun main(){
    val sim = Model("MM1 Model")
    sim.numberOfReplications = 30
    sim.lengthOfReplication = 20000.0
    sim.lengthOfReplicationWarmUp = 5000.0
    SimpleServiceSystem(sim, 1, name = "MM1")
    sim.simulate()
    sim.print()
}
```

The following represent some basic results from running the model. In addition to text-based output, the results can be saved to comma separated value files, data frames, a database and other storage and reporting frameworks. Notice that useful statistics are automatically collected on the resource and its queue.

```
Half-Width Statistical Summary Report - Confidence Level (95.000)%
Name                    Count          Average        Half-Width
-----------------------------------------------------------------------------
Servers:InstantaneousUtil    30          0.4988          0.0017
Servers:NumBusyUnits         30          0.4988          0.0017
Servers:ScheduledUtil        30          0.4988          0.0017
```

Servers:Q:NumInQ	30	0.4945	0.0083
Servers:Q:TimeInQ	30	0.4950	0.0079
MM1:NumInSystem	30	0.9932	0.0093
MM1:TimeInSystem	30	0.9944	0.0085
Servers:SeizeCount	30	14982.0333	40.9728
MM1:NumServed	30	14981.7333	41.0323

The purpose of this section was to provide a quick overview of the event view and process view modeling constructs available in the KSL. For extensive details on these constructs, including their implementation, the interested reader should consult Rossetti (2023a). Building on the ideas of this section, the following sections will illustrate KSL modeling for a situation suitable for a tutorial presentation.

3 EXAMPLE APPLICATION MODEL DEVELOPMENT

This section presents a small example to illustrate the modeling capabilities of the KSL. The purpose is to solve a small (but interesting) simulation problem and provide a tutorial on the KSL's modeling approach. Consider the following design problem.

3.1 Health Clinic Example

As part of a diabetes prevention program, a clinic is considering setting up a screening service in a local mall. They are considering two designs involving how the workers in the clinic provide service.

Design A: After waiting in a single line, each walk-in patient is served by one of three available nurses. Each nurse has their own booth, where the patient is first asked some medical health questions, then the patient's blood pressure and vitals are taken, finally, a glucose test is performed to check for diabetes. In this design, each nurse performs the tasks in sequence for the patient. If the glucose test indicates a chance of diabetes, the patient is sent to a separate clerk to schedule a follow-up at the clinic. If the test is not positive, then the patient departs.

Design B: After waiting in a single line, each walk-in is served in order by a clerk who takes the patient's health information, a nurse who takes the patient's blood pressure and vitals, and another nurse who performs the diabetes test. If the glucose test indicates a chance of diabetes, the patient is sent to a separate clerk to schedule a follow-up at the clinic. If the test is not positive, then the patient departs. In this configuration, there is no room for a patient to wait between the tasks; therefore, a patient who has had their health information taken cannot move ahead unless the nurse taking the vital signs is available. Also, a patient having their glucose tested must leave that station before the patient in blood pressure and vital checking can move ahead.

Patients arrive to the system according to a Poisson arrival process at a rate of 9.5 per hour (stream 1). Assume that there is a 5% chance (stream 2) that the glucose test will be positive. For design A, the time that it takes to have the paperwork completed, the vitals taken, and the glucose tested are all log-normally distributed with means of 6.5, 6.0, and 5.5 minutes, respectively (streams 3, 4, 5). They all have a standard deviation of approximately 0.5 minutes. For design B, because of the specialization of the tasks, it is expected that the mean of the task times will decrease by 10%.

Assume that the clinic is open from 10 am to 8 pm (10 hours each day) and that any patients in the clinic before 8 pm are still served. The distribution used to model the time that it takes to schedule a follow up visit is a Weibull distribution with shape 2.6 and scale 7.3 that should use stream 6.

Make a statistically valid recommendation as to the best design based on the average system time of the patients. We want to be 95% confident of our recommendation to within 2 minutes.

3.2 Solution for Design A

We will use the process view to model this situation. Design configuration A for the clinic is essentially a multi-server system (3 nurses) followed by a single-server system (scheduling clerk). Thus, the form of this model will be very similar to the previously illustrated M/M/1 model. The main difference is the delay time to perform the three tasks (paperwork, vitals, and glucose testing). Then, the chance that the patient needs to schedule a follow up visit must be modeled. The other issue that must be handled (for both design configurations) is that the clinic is open for 10 hours and does not accept new patients after that time; however, any patients already in the clinic at the end of 10 hours are processed until there are no patients. This indicates that the patient arrival process must be turned off after 10 hours, but the processing needs to continue. Thus, the simulation run length is not known in advance.

To model design A, we need to define the random variables and instrument the model for statistical collection. The following code presents these definitions.

```
class ClinicDesignA(parent: ModelElement,name: String? = null) :
ProcessModel(parent, name) {

    private var timeBetweenArrivals: RandomVariable = RandomVariable(parent,
        ExponentialRV(60.0 / 9.5, 1))

    private var needsFollowUp: RandomVariable = RandomVariable(this,
        BernoulliRV(0.05, 2))

    private var paperWorkTime: RandomVariable = RandomVariable(this,
        LognormalRV(6.5, 0.5 * 0.5, 3))

    private var vitalsTime: RandomVariable = RandomVariable(this,
        LognormalRV(6.0, 0.5 * 0.5, 4))

    private var diabetesTestTime: RandomVariable = RandomVariable(this,
        LognormalRV(5.5, 0.5 * 0.5, 5))

    private var schedulingTime: RandomVariable = RandomVariable(this,
        WeibullRV(2.6, 7.3, 6))

    private val wip: TWResponse = TWResponse(this,
        "${this.name}:NumInSystem")

    private val timeInSystem: Response = Response(this,
        "${this.name}:TimeInSystem")
    val systemTime: ResponseCIfc
        get() = timeInSystem

    private val numPatients: Counter = Counter(this,
        "${this.name}:NumPatients")
```

The time between arrivals is modeled with an exponential random variable and whether the patient needs to schedule a follow up is modeled with a Bernoulli random variable with the specified probability of success of 0.05. The paperwork time, time to take the vitals, and the diabetes testing time are all lognormal random variables with the parameters specified in the problem statement. The scheduling time

is governed by a Weibull random variable. The specifying of the input distributions is a necessary and common aspect of any simulation model.

As was done in the example M/M/1 model, the number of patients in the system is captured with a *TWResponse* variable, the system time is captured with a *Response* variable, and a counter is used to tabulate the total number of patients processed. For the purposes of this example, the system time is required to make the comparison between the two designs. Notice that the code exposes the underlying response by defining a public property (*systemTime*) of type *ResponseCIfc* interface. The *ResponseCIfc* interface defines a read-only view of a response to facilitate reporting of statistics. Its use will be illustrated in section 4. Now we are ready to model the clinic's process.

To begin the process modeling, we need to define the resources that will be used and then model the entities that use the resources. In the following code, the variable *nurses* is defined as a resource with a queue, with the number of nurses (capacity) set to 3 units. This represents that any of the nurses can perform the paperwork, vitals, and diabetes testing. The scheduling clerk is a separate resource of capacity 1 to handle the follow-up scheduling. Before proceeding with how to generate the patients, let's review the patient's process.

```
private val nurses: ResourceWithQ = ResourceWithQ(this,
    "Nurses", capacity = 3)

private val schedulingClerk: ResourceWithQ = ResourceWithQ(this,
    "SchedulingClerk", capacity = 1)

private val patientGenerator = EntityGenerator(
    ::Patient, timeUntilTheFirstEntity = timeBetweenArrivals,
    timeBtwEvents = timeBetweenArrivals,
    timeOfTheLastEvent = 10.0 * 60.0
)

private inner class Patient : Entity() {
    val patientProcess: KSLProcess = process() {
        wip.increment()
        timeStamp = time
        val nurse = seize(nurses)
        delay(paperWorkTime)
        delay(vitalsTime)
        delay(diabetesTestTime)
        release(nurse)
        if (needsFollowUp.value == 1.0) {
            use(schedulingClerk, delayDuration = schedulingTime)
        }
        timeInSystem.value = time - timeStamp
        wip.decrement()
        numPatients.increment()
    }
}
```

The inner class, *Patient*, sub-classes from *Entity* and defines the process that the patient experiences within the clinic. Similar to the M/M/1 example, when the patient's process is activated the work in process is incremented and the time of arrival is time-stamped. Then, we have the key aspect of this model. First a

nurse is seized, then the three task times occur, each with a *delay()* to indicate the time, and finally the nurse is released.

If the Bernoulli random variable is a 1, then the patient needs to visit the scheduling clerk. Notice that the standard flow of control constructs (e.g. if-else, while, for, etc.) are available for use in implementing the process. The second thing to notice is the KSL process, *use*() function. Because the pattern of seize-delay-release is so common, the KSL provides a suspending function that encapsulates this pattern. To use a resource, specify both the resource being used and the time of use. At the end of a patient's process, we see that the time spent in the system is tabulated (which may include the scheduling activity). Then the work in process is decremented and the number of patients processed is incremented before the process ends.

Different from the code presented for the M/M/1 example, this code illustrates a new approach to creating and activating the patient entities. In this example code, the creation pattern was implemented via an *EntityGenerator*. An instance of the *EntityGenerator* class defines a creation pattern for entities that is similar to the create or source constructs found in many simulation languages. The generator requires the time until the first creation, the time between creations, and a function that will create the entity. In the code the parameter (*::Patient*) refers to the constructor of the Patient entity. In addition, an entity generator can specify a maximum number of creation events and the time of the last creation event. If the time of the last event is specified, then when the event time occurs the generator turns itself off. Because the clinic is open for 10 hours but does not permit new patients after that time, the specification of this parameter is all that is necessary to turn the patient creation process off. Any patients that are in process after 10 hours (600 minutes) will continue to execute their processes to completion, but no new patients will be created after this time.

The results of running this model are illustrated in section 4. In the next sub-section, we present the modeling of design B.

3.3 Solution for Design B

Rather than repeat the code to define the random variables and responses, this section emphasizes how to model the requirement of this new design which entails no waiting space between the tasks. The implementation of the random variables is exactly as previously presented except that the mean task time is reduced by 10%. However, the resources and process for this design is different. The following code defines the two clerks for paperwork and post-test scheduling. In design B, there are only two nurses, one dedicated to taking the patient's vitals and the second nurse dedicated to the glucose testing.

```
private val schedulingClerk: ResourceWithQ = ResourceWithQ(this,
    "SchedulingClerk", capacity = 1)

private val paperWorkClerk: ResourceWithQ = ResourceWithQ(this,
    "Paperwork Clerk", capacity = 1)

private val vitalsNurse: ResourceWithQ = ResourceWithQ(this,
    "Vitals Nurse", capacity = 1)

private val testNurse: ResourceWithQ = ResourceWithQ(this,
    "Glucose Test Nurse", capacity = 1)

private inner class Patient : Entity() {
    val patientProcess: KSLProcess = process() {
        wip.increment()
        timeStamp = time
        val c1 = seize(paperWorkClerk)
```

```
            delay(paperWorkTime)
            val n1 = seize(vitalsNurse)
            release(c1)
            delay(vitalsTime)
            val n2 = seize(testNurse)
            release(n1)
            delay(diabetesTestTime)
            release(n2)
            if (needsFollowUp.value == 1.0) {
                use(schedulingClerk, delayDuration = schedulingTime)
            }
            timeInSystem.value = time - timeStamp
            wip.decrement()
            numPatients.increment()
        }
    }
```

The key to modeling the lack of space between the tasks is to not release the previous resource until the next resource is allocated. In the above code, we see that the paperwork clerk is seized, and the associated delay occurs; however, after the delay, the clerk is not immediately released. Instead, the patient attempts to seize the vitals nurse. If the vitals nurse is not available, the patient waits in the vital nurse's queue and does not release the paperwork clerk. This prevents arriving patients from getting the paperwork clerk until patient can start the vitals taking activity after the vitals nurse is available. If the patient succeeds in getting the vitals nurse, then the patient releases the paperwork clerk. Thus, the patient cannot move forward in the process until the next resource is available. These concepts are illustrated in the activity diagram presented in Figure 1.

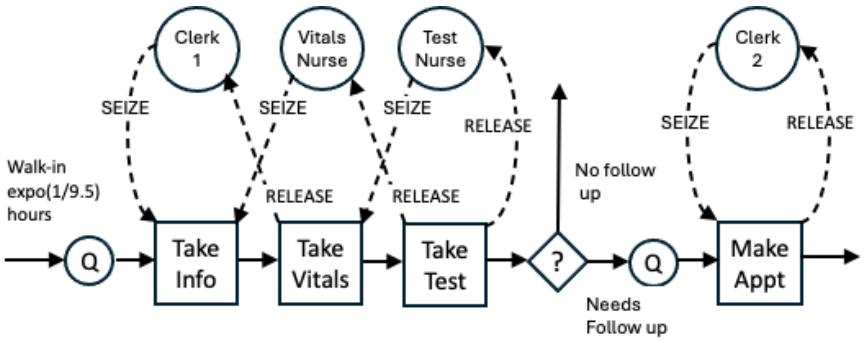

Figure 1: Activity Diagram for Design B Process

Notice in Figure 1 that the seizing of the next resource overlaps with the release of the previous resource. The patient attempts to seize the next resource, and if the resource is available the patient proceeds. If the resource is not available, then the patient stays at the previous location. This is an example of a tandem queue with blocking.

4 ANALYSIS FOR HEALTH CLINIC EXAMPLE

This section illustrates the setup, execution, and analysis of the previously described models for design A and B for the health clinic problem. The KSL has extensive functionality for capturing and analyzing data generated from simulation models. The basic functionality includes:

- automatically capturing within and across replication results to comma separated value files,
- capturing replication data via the *ReplicationDataCollector* class,
- tracing a response variable via the *ResponseTrace* class,
- using the *SimulationReporter* class to output results in text or MarkDown and adjusting the confidence level of the reports,
- capturing all simulation results to a database using the *KSLDatabaseObserver* class.

To keep the presentation simple and brief, this presentation illustrates the use of the *ReplicationDataCollector* class. The *ReplicationDataCollector* class is an observer that can be attached to a model that extracts replication summary data to arrays for in-memory post-processing. This example uses instances of the *ReplicationDataCollector* class to capture the patient's average system time for each replication for both of the models. The following code illustrates the running of the models for a set of pilot simulations to plan the necessary number of replications.

```
fun main(){
    val modelA = Model("Clinic A Model")
    val clinicA = ClinicDesignA(modelA, "Design A")
    modelA.numberOfReplications = 10

    val rdc1 = ReplicationDataCollector(modelA)
    rdc1.addResponse(clinicA.systemTime)
    modelA.simulate()
    modelA.print()

    val modelB = Model("Clinic B Model")
    val clinicB = ClinicDesignB(modelB, "Design B")
    modelB.resetStartStream()
    modelB.numberOfReplications = 10

    val rdc2 = ReplicationDataCollector(modelB)
    rdc2.addResponse(clinicB.systemTime)

    modelB.simulate()
    modelB.print()

    val dataA = rdc1.replicationData(clinicA.systemTime.name)
    val dataB = rdc2.replicationData(clinicB.systemTime.name)

    val dataMap = mutableMapOf(
        "Design A System Time" to dataA, "Design B System Time" to dataB )

    val mc = MultipleComparisonAnalyzer(dataMap)

    println(mc)

}
```

As previously illustrated, each model is created, the number of replications specified, and the model simulated. The *ReplicationDataCollector* class instances are attached to the model prior to executing the

replications and the desired response (*systemTime*) is added to the data collector. Then, the data is extracted from the data collector with the lines:

```
val dataA = rdc1.replicationData(clinicA.systemTime.name)
val dataB = rdc2.replicationData(clinicB.systemTime.name)
```

Then, a data map is constructed for use with the KSL's analysis class, *MultipleComparisonAnalyzer*. The *MultipleComparisonAnalyzer* class will compute all pairwise differences for the data in the supplied data map and compute statistics on the data. The following console output is a portion of the output produced by the *MultipleComparisonAnalyzer* class. Based on this pilot run, we can suspect that design B has a longer system time for the patients; however, since the confidence interval on the difference contains zero, we cannot reject the hypothesis that the system times are equal. In essence, additional replications are required to decide between the two design configurations.

```
Statistical Summary Report - Raw Data
Name                  Count        Average          Std. Dev.
-------------------------------------------------------------------
Design A System Time      10        34.6257             9.8472
Design B System Time      10        40.2726            17.3669
-------------------------------------------------------------------

95.0% Confidence Intervals on Data
Design A System Time  [27.581444876767538, 41.66991964508021]
Design B System Time  [27.849054719812024, 52.69612952546536]

Statistical Summary Report - Difference Data
Name                                  Count        Average      Std. Dev.
-------------------------------------------------------------------------------
Design A System Time - Design B System Time  10       -5.6469      23.4420
-------------------------------------------------------------------------------

95.0% Confidence Intervals on Difference Data
Design A System Time - Design B System Time     [-22.41633159106079, 11.122511867631168]
```

Based on standard deviation of the difference (23.442) and the requirement that we want to be 95% confidence that the true difference is within plus or minus 2 minutes, we can approximate the required sample size using the following formula.

$$n \geq \left(\frac{z_{1-\left(\frac{\alpha}{2}\right)}S}{\varepsilon}\right)^2 = \left(\frac{1.9599 \times 23.442}{2}\right)^2 = 527.7$$

Rerunning the simulation for 528 replications produces the following results. By the way, the KSL *Statistic* class has functions implementing this and other sample size calculations.

```
Statistical Summary Report - Raw Data
Name                  Count        Average          Std. Dev.
-------------------------------------------------------------------------------
Design A System Time      528       40.3496            16.3535
Design B System Time      528       37.8736            14.4309
```

```
--------------------------------------------------------------------------------
95.0% Confidence Intervals on Data
Design A System Time       [38.95154341157163, 41.74775569525546]
Design B System Time[36.639875021755536, 39.107344748779205]

Statistical Summary Report - Difference Data
Name                                Count     Average   Std. Dev.
-----------------------------------------------------------------------------
Design A System Time - Design B System Time    528         2.4760            21.2616
-----------------------------------------------------------------------------
95.0% Confidence Intervals on Difference Data
Design A System Time - Design B System Time     [0.6583243607943041, 4.293754975498095]
```

Based on these results, we can be 95% confident that design configuration B has a lower system time, which is not what we would have expected based on the initial pilot run.

5 SUMMARY AND FUTURE WORK

The KSL provides an application programming interface (API) for developing and executing discrete-event simulation models within the Kotlin programming language.

This tutorial illustrated some of the fundamental constructs of the KSL, including how to build model elements that can simulate using the event view or the process view. The KSL has classes that facilitate the modeling of queues, resources, activities, and processes found in many simulation languages. In addition to the constructs illustrated in this tutorial, the KSL has constructs that facilitate:

- piecewise constant and piecewise linear modeling of non-stationary Poisson processes,
- FIFO, LIFO, Random, Prioritized queues,
- resources that have time varying capacity,
- schedule based collection of statistics as noted in Smith and Nelson (2015),
- aggregation of statistics, dynamic histograms, statistical batching, historical (trace-driven) variables,
- automatic distribution fitting and recommendation for input models,
- instrumentation of control variables for large-scale simulation experiments, and
- construction of commonly needed plots in simulation such as Welch plots, partial sums plots, ACF plots, Q-Q plots, P-P plots, and state variable plots.

Finally, the development process currently involves implementations that will bring the following to KSL models:

- mobile resources (transporters),
- conveyors, and
- automated guided vehicles.

While the KSL does not currently offer animation, the feasibility of providing animation is currently under investigation. It is hoped that the open-source, freely available nature of the KSL will enable more applications of simulation within engineering, operations research, and data science workflows.

REFERENCES

Law, A. 2007. "Simulation Modeling and Analysis", 4th Ed., McGraw-Hill.

Loftus, R. 2022. "The 15 Most Popular Programming Languages of 2023". *Hacker Rank*, Accessed, April 5th, 2023, https://www.hackerrank.com/blog/most-popular-languages-2023/.

Rossetti, M. D. 2023a. Simulation Modeling using the Kotlin Simulation Library (KSL), Open Text Edition. Retrieved from https://rossetti.github.io/KSLBook/ licensed under the Creative Commons Attribution-Noncommercial-No Derivatives 4.0 International License.

Rossetti, M. D. 2023b. "KSL Open-Source Repository", Accessed April 5th, 2024, https://rossetti.github.io/KSLDocs/-k-s-l-core/ksl.utilities/index.html.

Rossetti, M. D. 2023c. "KSL API Documentation", Accessed April 5th, 2024, https://rossetti.github.io/KSLDocs/index.html.

Rossetti, M. D. 2023d. "Introducing the Kotlin Simulation Library (KSL)". In 2023 Winter Simulation Conference (WSC), 3311-3322, https://dl.acm.org/doi/10.5555/3643142.3643418.

Smith, J. and B. L. Nelson. 2015. "Estimating and Interpreting the Waiting Time for Customers Arriving to a Non-Stationary Queueing System". In 2015 Winter Simulation Conference (WSC), 2610–2621, https://10.1109/WSC.2015.7408369

AUTHOR BIOGRAPHY

MANUEL D. ROSSETTI is a University Professor of Industrial Engineering and the Director of the Data Science Program at the University of Arkansas. He received his Ph.D. in Industrial and Systems Engineering from The Ohio State University. Previously, he served as the Associate Department Head for the Department of Industrial Engineering and as the Director for the NSF I/UCRC Center for Excellence in Logistics and Distribution (CELDi) at UA. Dr. Rossetti has published two textbooks and over 125 refereed journal and conference articles in the areas of simulation, logistics/inventory, and healthcare and has been the PI or Co-PI on funded research projects totaling over 6.5 million dollars. In 2013, Rossetti received the Charles and Nadine Baum Teaching Award, the highest teaching honor bestowed at the UofA, and was elected to the UofA's Teaching Academy. Dr. Rossetti was elected as a Fellow for IISE in 2012, in 2021 received the IISE Innovation in Education competition award, and in 2023 he received the Albert G. Holzman Distinguished Educator Award. He serves as an Associate Editor for the International Journal of Modeling and Simulation and is active in IISE, INFORMS, and ASEE. He served as co-editor for the WSC 2004 and 2009 conference, the Publicity Chair for the WSC 2013 Conference, 2015 WSC Program Chair, and is the General Chair for the WSC 2024. He can be contacted at rossetti@uark.edu and https://rossetti.uark.edu/.