Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

INTRODUCING THE KOTLIN SIMULATION LIBRARY (KSL)

Manuel D. Rossetti

University of Arkansas
Department of Industrial Engineering
4207 Bell Engineering Center
Fayetteville, AR, 72701, USA

ABSTRACT

This paper introduces a Monte Carlo and discrete-event simulation library for the Kotlin programming language. The Kotlin Simulation Library (KSL) provides functionality to perform simulation experiments involving the generation of random processes, the execution of discrete-event simulation via the event and process views, and the analysis of the statistical quantities generated by simulation models. The architecture of the library leverages the object-oriented and functional programming capabilities of the widely used Kotlin programming language. The library provides functionality that is similar to proprietary software, while being open-source and readily extensible. This paper provides an overview of the architecture of the library. The functionality of the library is illustrated through several examples.

1 INTRODUCTION

This paper introduces the Kotlin Simulation Library for the Kotlin programming language. Kotlin is a widely available, open source, general purpose programming language introduced in 2011 by JetBrains. Due to its history, the language originally targeted the Java virtual machine (JVM); however, the language has evolved to include support for multi-platforms (JVM, java script (JS), and native). Developers can develop multi-platform libraries with common code and platform specific implementations for JVM, JS, and native platforms. 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 since its arrival. In fact, according to Joslyn and Hecht (2023), Kotlin is expected to get the 3rd largest increase (of 8%) in new users in 2023 behind Go and Rust, and ahead of Python.

Kotlin's popularity is likely driven by two factors: 1) it is 100% compatible with the Java ecosystem, and 2) it was designed from the ground up to be developer friendly. The fact that Java has been ranked as the number 1 programming language for many years and is likely to continue to do so for the foreseeable future bodes well for the continuing and increasing popularity of Kotlin. For Java developers, Kotlin is an extremely easy language to learn. Kotlin was designed to overcome many of the flaws and issues with the design of the Java programming language, including for example, an explicit treatment of null references. In addition, since Kotlin is 100% compatible with the Java ecosystem, developers can utilize the many existing proprietary and open-source libraries that are available to Java programmers. Kotlin is less verbose than many languages, especially Java, has a modern syntax with static typing, and supports both object-oriented and functional programming. Of primary interest to the simulation community is the fact that Kotlin has first-class support for *coroutines*, which had been lacking for the Java programming language until 2022. The support for coroutines facilitates the implementation of the process-view of simulation, which will be illustrated in subsequent sections.

The rest of this paper is structured as follows. The following section will provide some more background on simulation libraries and review some of the literature in that area. Then, section 3 will

overview the architecture and functionality of the KSL, including a discussion of the basic library support for random variate generation, statistical collection, discrete-event modeling, and various utilities. Then, section 4 will illustrate the use of the KSL via some examples within the Monte-Carlo domain and within discrete-event modeling.

2 BACKGROUND AND LITERATURE REVIEW

Attempting to implement simulation models, from scratch, in a general-purpose language such as FORTRAN, Visual Basic, C/C++, Java, and Kotlin requires above average programming skills. In the absence of specialized libraries for these languages that try to relieve the user from some of the burden, simulation as a tool would be relegated to "elite" programmers. The repetitive nature of computations in simulation allows the development of computer libraries that are applicable to simulation modeling situations. Thus, over the years a wide variety of simulation libraries have been developed for various available computer languages. One such library is the Java Simulation Library (JSL) discussed in Rossetti (2008). The JSL has been applied to transportation, logistics, manufacturing, and healthcare systems resulting in at least 5 journal papers and over 10 conference papers. Peixoto et al. (2017) built upon the JSL to construct URURAU to provide a graphical user interface and extend the library for process-oriented models. Dagkakis and Heavey (2016) provide criteria and a methodology for reviewing open-source simulation tools. They reviewed over 42 different discrete-event simulation tools and highlight the level of functionality, language, documentation, as well as a number of other characteristics. Since the popularity of the Python language has grown, the SimPy library (Team SimPy, 2017) for simulation has gained increased visibility including the development of specialized constructs for manufacturing simulation, see for example Olaitana et al. (2014) and Dagkakis et al. (2015). The interested reader is referred to Dagkakis and Heavey (2016) for a detailed review of various open-source simulation language implementations.

The interest in simulation libraries has remained steady as new languages are developed. An important aspect of modern programming languages has finally caught up to the requirements of discrete-event simulation libraries: *coroutines*. According to Wikipedia, a coroutine is a "computer program component that allows execution to be suspended and resumed, generalizing subroutines for cooperative multi-tasking", see (de Moura and Ierusalimschy, 2004). When Java first appeared with its support for threads, there was a lot of activity around how to utilize threads to map process modeling for use in simulation, see Rossetti (2000). Unfortunately, the cost of context switching of threads within an operating system makes implementing the process-view of simulation via the standard Java thread model computationally impractical. Since that time, much effort has been expended to add coroutine support to Java, see Weatherly and Page (2004) and Stadler et al. (2010), with finally a preview release in Java 19 in 2023.

In a sense, coroutines can be thought of as very light-weight threads. When a language has coroutines (or continuations), the basic underlying functionality of suspending and resuming code permits the implementation of the process view of simulation. The KSL takes advantage of Kotlin's coroutines library (Breslav and Elizarov, 2023) to provide the process view. Kotlin coroutines are implemented as part of a language library by mapping continuations on a generated state machine. The key to understanding Kotlin coroutines is to understand that coroutines utilize continuations (saved state) within a context. Functions that use coroutines are called suspending functions and are marked with the keyword *suspend*. The general use case for coroutines is to provide asynchronous programming constructs. Within the KSL, the coroutine continuation is used within the same execution thread to implement suspendable functions that provide the process view. A detailed discussion of the implementation of the process view for the KSL is provided in Chapter 6 of Rossetti (2023a). This paper focuses on the capabilities that are available within the KSL rather than the implementation details. In the next section, the functionality of the KSL is discussed.

3 OVERVIEW OF ARCHITECTURE AND FUNCTIONALITY

The KSL provides functionality that facilitates both Monte-Carlo (static) simulation and discrete-event dynamic system (DEDS) simulation. As a software library, the functionality is organized into various

packages (and sub-packages) that hold artifacts involving common modeling concepts and that provide separate programming namespaces. The main packages of the KSL include:

- *ksl.utilities* – This package provides the application programming interface (API) for classes that are useful across the entire library. This includes support for probability distribution modeling, random variate generation, statistical computations, specialized mathematical functions, and input/output utilities. This package primarily supports Monte-Carlo simulation activities.
- *ksl.simulation* – This package primarily supports the running of DEDS simulation models. The package contains the classes that must be instantiated to build and run discrete-event models. The most important classes within this package are the *Model* class and the *ModelElement* class, which are used to construct DEDS models.
- *ksl.modeling* – This package contains sub-classes of the *ModelElement* class. These are model building artifacts such as queues, resources, random variables, entities, processes, and response variables. These classes are used to define and represent the physical and conceptual components of system in the form of a model. Because of the object-oriented nature of these elements, library users can extend and create new model elements to represent systems that are being modeled.
- *ksl.calendar* – This package contains the data structures used to represent the event calendar used within DEDS modeling. Generally, library users do not work directly with these classes.
- *ksl.controls* – The controls package supports the execution of DEDS simulation experiments by allowing modelers to define properties within their models that can be controlled (changed) in a general manner across a variety of experimental setting. The controls package is under active development with the intention to support experiments that execute within a software as service using cloud platforms.
- *ksl.observers* – The observers package provides classes that can be attached to model elements with the intention of observing, reacting, or capturing state changes over time. These classes allow the library user to instrument their model to trace, collect, and react to observed behavior over time.

Each of the primary packages are sub-divided into sub-packages that organize the classes needed to support the purpose of the main package and of the KSL as a whole. The interested reader is referred to Rossetti (2023a) for further details to access the KSL open-source repository and the KSL API documentation. The following will provide an overview of the most important functionality.

3.1 Monte-Carlo Simulation Via the *ksl.utilities* Package

The KSL utilities package has an extensive set of components that support the development and execution of Monte-Carlo (MC) simulation experiments. Within the *utilities* package, the KSL separates into two sub-packages the concerns of probability modeling (*ksl.utilities.distributions*) and generating random variates (*ksl.utilities.random*). For the purposes of this paper, the main package of interest is the *ksl.utilities.random* package, which supports the modeling of randomness within simulation models.

Within the KSL modeling paradigm, probability distributions represent how to compute quantities related to the probability model, but do not directly support the generation of random variates. Within the KSL, the concept of a random variable, is generalized in such a manner such that any random process can result in random variates (rather than being limited to coming from specific distributions). This abstraction also relieves the necessity of dealing with and supporting some of the computational complexities required in probability distribution modeling. For example, while it may be difficult to compute the moments and loss functions of a probability distribution, these requirements are not typically necessary to *generate* from the distribution. While still supporting probability computations concerning many of the common univariate distributions, the main focus of the KSL is on generating random quantities via the *ksl.utilities.random* package. Of particular note, is the *ksl.utilities.random.rvariable* package.

Figure 1 presents the main interfaces within the KSL that support the generalized concept of randomness. Things that generate random quantities tend to implement these interfaces.

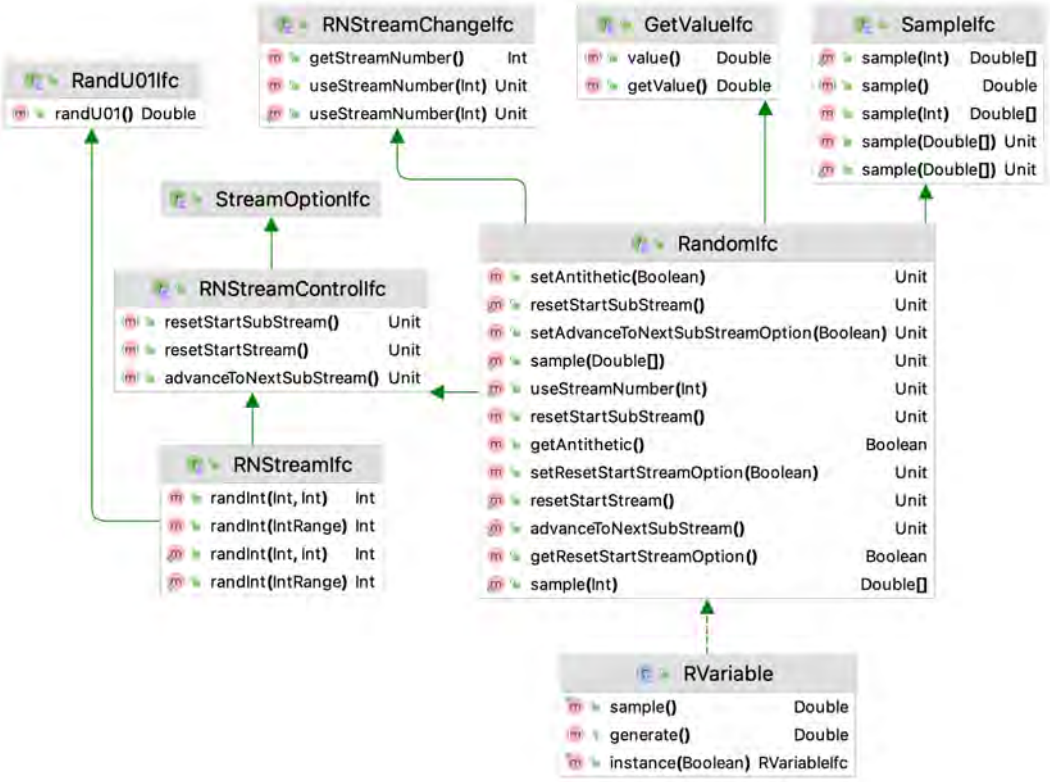

Figure 1: Main interfaces supporting randomness.

As can be noted in Figure 1, randomness is supported via the use of random number streams. A random number stream is a sequence of pseudo-random U(0,1) numbers that also provides the capability to move within the stream as discussed in L'Ecuyer et al. (2002). Sampling from the source of randomness is facilitated via the *SampleIfc* interface, which relies on the *GetValueIfc* interface to provide the randomly generated value. The abstract base class, *RVariable*, provides a starting point for implementing the functionality of random variables through its sole abstract function, *generate()*. Thus, users that need to implement their own random variables need only implement this function. The KSL supports all the distributions mentioned in Chapter 8 of Law (2007) plus many others.

Besides implementing these common random variables, the KSL supports the generation from mixture distributions, provide base functionality for implementing the inverse transform technique, acceptance-rejection, and the ratio of uniforms algorithms, and supports the generation from multi-variate distributions. In addition, the KSL provides for a random variable algebra that allows the definition of new random variables as functionals of existing random variables. This capability will be illustrated in the next section.

Besides the generation of random quantities, the KSL *ksl.utilities.statistic* package provides functionality to observe and compute a wide variety of statistical quantities. This includes summary statistics across observations (including lag 1 correlation), box plot summaries, histograms summaries, frequency tabulation, weighted statistics, bootstrap statistics, and multiple comparison statistics. Due to space limitations, this functionality is not discussed in this paper; however, the interested reader is referred to Rossetti (2023a) for further details. This functionality is illustrated in examples within the next section.

Finally, the *utilities* package has general purpose functionality to improve simulation developer's productivity by providing simplified abstractions for working with files, file directories, comma separated value files, Excel files, Markdown output, and databases. In addition, by using the Kotlin language's *extension* function capabilities, the KSL provides an extremely useful set of utilities for working with 1-D

and 2-D arrays such as reading, writing, element wise addition, subtraction, multiplication, division, applying functionals to elements, matric multiplication, searching, converting, counting, and checking elements, etc. In many cases, this functionality has been defined as an extension function, which then permits using the array as the base unit of operation. This capability will be illustrated in a subsequent example.

3.2 DEDS Support Via the *ksl.modeling* and *ksl.simulation* Packages

The KSL *modeling* and *simulation* packages provide the primary functionality for developing and executing DEDS simulation models. Figure 2 provides an overview of some of the many classes available within the modeling and simulation packages of the KSL. These classes provide important abstractions for representing system components within a KSL model. While it is beyond the scope of this article to provide a detailed discussion of each of these classes, a basic overview of the major classes will be mentioned here. The interested reader should refer to Rossetti (2023a) for further details.

The most important class for DEDS modeling is the *ModelElement* abstract base class. To model system components, users must sub-class the *ModelElement* class to represent the system properties and behaviors. The *ModelElement* class provides the following capabilities:

- ability to schedule events onto the simulation event calendar
- ability to hold instances of model elements that represent the system components
- a well-defined pattern for automated actions that occur during simulation execution, which includes initializing the state of the system, warming up the statistical collectors, capturing final system state/data, clearing within replication statistical accumulators, capturing across replication statistical summaries, and cleaning up after replications

Instances of sub-classes of *ModelElement* form KSL models. Figure 2 shows many of the existing concrete instances of *ModelElement* available within the KSL. For example, the *Queue* class provides the ability to hold instances of class *QObject* such that statistics on the usage of the queue are automatically collected. Within Figure 2, there is a hierarchy of classes starting with the *Variable* class that represent types of variables that can be used within a model including variables to collect statistics on tally type variables (*Response*) and time-weighted variables (*TWResponse*). The KSL provides the ability to collect aggregates of existing response variables by observing their changes. For example, suppose a system had five queues and the user wanted to collect statistics on the total number waiting in all the queues at any time. The KSL provides the *AggregateTWResponse* class that, after being attached to the queues, will automatically track the aggregate response and its statistics. In addition, the KSL provides the capability to capture statistics by periods of time via a schedule. For example, in the case of non-stationary system modeling, the performance during specific intervals of time, e.g. the peak period, should be reported to fully understand system behavior. Smith and Nelson (2015) describe this modeling issue and discuss its importance for system modeling.

As noted in Figure 2, the KSL also defines constructs for generating events according to creational patterns, *EventGenerator*, and according to non-stationary Poisson processes (*NHPPEventGenerator*). The use of the *RandomVariable* class facilitates the management of random number streams to facilitate the usage of common random numbers as a variance reduction technique.

The KSL also provides the ability to model a system via both the event and the process views. Due to paper length limitations, a complete discussion of the event-view is not presented; however, the interested reader can refer to Chapters 4 and 5 of Rossetti (2023a) for further details. The process view allows the modeler to describe the flow of the entities within a system via the entity's view. In the process view, the modeler imagines that they are the entity of interest (e.g. part) and describes the interaction with the system via a domain specific language. Proprietary commercial simulation software often utilizes this modeling paradigm because it is simpler for users. The KSL provides constructs for modeling entities and describing their processes. The *ProcessModel* class provides the ability to create entities, define processes, utilize

resource, and model movement. The support for process modeling includes the following classes: *HoldQueue* (for suspending a process while the entity is within a queue), *Signal* (for holding and signaling suspended entities), *Resource* (for interacting with entities for quantities needed before proceeding), *MobileResource* (for transporter like modeling), *EntityGenerator* (creating entities according to a time-based pattern), and *BlockingQueue* (for communicating between processes). For further details on the implementation of process modeling within the KSL, the reader should refer to Chapter 6 of Rossetti (2023a). An example in the following section will illustrate and discuss some of the possibilities of using the KSL for process modeling.

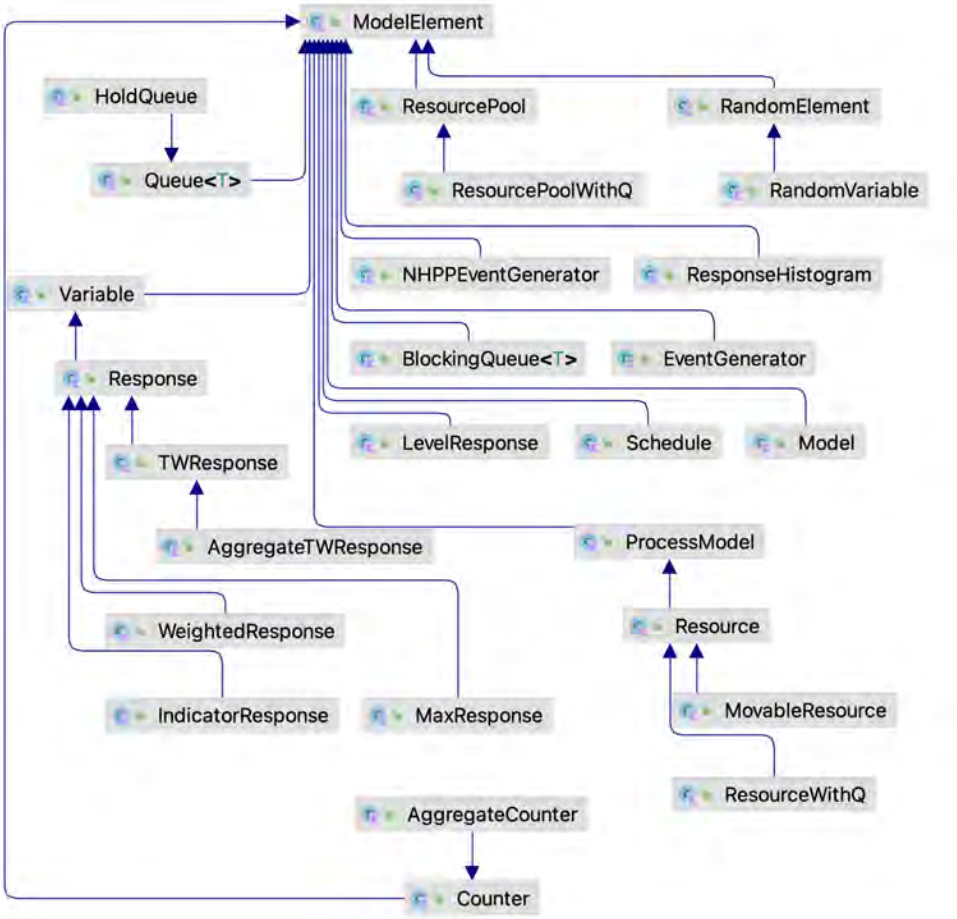

Figure 2: Example classes for modeling within the KSL.

4 ILLUSTRATIVE EXAMPLES

This section provides examples of the use of the KSL. The first example illustrates the use of KSL constructs for MC simulation. In the following example line 4 creates an instance of a random number stream that references the first stream in the sequence of streams from the underlying generator. Line 6 creates a separate duplicate instance of the same stream that has the same underlying state. Lines 7 and 8 create normal random variables that use the streams. Lines 9 and 10 sample from the random variables and store the samples in arrays. The pairwise difference is performed using the *KSLArrays* utilities in line 11. In line 12, the extension function for Double arrays writes the differences out to a text file, and in line 13 another extension function computes summary statistics on the values in the array.

```
01    fun example1(){
02        // illustrate common random numbers
03        // normals are dependent, because same stream
04        val n1Stream = KSLRandom.rnStream(1)
05        // n2Stream is a clone of n1Stream but not the same object
06        val n2Stream = n1Stream.instance()
07        val n1 = NormalRV(2.0, 0.64, n1Stream)
08        val n2 = NormalRV(2.2, 0.36, n2Stream)
09        val s1 = n1.sample(100)
10        val s2 = n2.sample(100)
11        val d = KSLArrays.subtractElements(s1, s2)
12        d.writeToFile("Differences")
13        val stats = d.statistics()
14        println(stats)
15        // compute statistics found on a box plot
16        val boxPlotSummary = BoxPlotSummary(d)
17        println(boxPlotSummary)
18        val breakPoints = Histogram.recommendBreakPoints(stats)
19        val h = Histogram(breakPoints)
20        h.collect(d)
21        println(h)
22    }
```

Line 16 illustrates how a box plot summary can be performed. In addition, lines 18-20 show how a histogram can be tabulated on the data in the array. This code produces a large amount of output via lines 14, 17, and 21. Below is a sample of the output. The summary produces standard statistical summaries, including a confidence interval, skewness, kurtosis, and lag-1 covariance and correlation.

```
Name Statistic_1
Number 100.0
Average -0.19201751883243698
Standard Deviation 0.1899701154208089
Standard Error 0.018997011542080892
Half-width 0.037694192034629904
Confidence Level 0.95
Confidence Interval [-0.22971171086706688, -0.15432332679780708]
Minimum -0.7751839644679286
Maximum 0.20189899277177847
Sum -19.2017518832437
Variance 0.03608864475299546
Deviation Sum of Squares 3.5727758305465507
Last value collected -0.059228909742418345
Kurtosis -0.2314557519668206
Skewness -0.28900946907387737
Lag 1 Covariance -0.0032778458472826682
Lag 1 Correlation -0.09174507449523456
Von Neumann Lag 1 Test Statistic -0.8229876116311975
Number of missing observations 0.0
Lead-Digit Rule(1) -2
```

The box plot summary applies the same algorithms as R to estimate the quartiles and fences associated with a boxplot. Finally, the histogram functionality will recommend reasonable breakpoints and then tabulate the frequencies associated with the bins. All of the quantities that are computed are available through the objects (boxplot, histogram, statistics) as either properties or functions so that the user can access the individual values that are needed. The *toString()* functions of the classes have been overloaded to provide useful and commonly needed output as illustrated here.

```
BoxPlotSummary
lower outer fence = -1.2014379519037894
lower inner fence = -0.7712716759403473
min = -0.7751839644679286
firstQuartile = -0.34110539997690514
median = -0.1776686381972259
thirdQuartile = -0.05432788266794375
max = 0.20189899277177847
upper inner fence = 0.37583839329549834
upper outer fence = 0.8060046692589404

Histogram: ID_6
-------------------------------------
Number of bins = 10
First bin starts at = -1.0
Last bin ends at = 1.0
Under flow count = 0.0
Over flow count = 0.0
Total bin count = 100.0
Total count = 100.0
-------------------------------------
Bin Range        Count CumTot Frac  CumFrac
  1 [-1.00,-0.80)  = 0   0.0 0.000000 0.000000
  2 [-0.80,-0.60)  = 1   1.0 0.010000 0.010000
  3 [-0.60,-0.40)  = 15  16.0 0.150000 0.160000
  4 [-0.40,-0.20)  = 27  43.0 0.270000 0.430000
  5 [-0.20,-0.00)  = 40  83.0 0.400000 0.830000
  6 [-0.00, 0.20)  = 16  99.0 0.160000 0.990000
  7 [ 0.20, 0.40)  = 1 100.0 0.010000 1.000000
  8 [ 0.40, 0.60)  = 0 100.0 0.000000 1.000000
  9 [ 0.60, 0.80)  = 0 100.0 0.000000 1.000000
 10 [ 0.80, 1.00)  = 0 100.0 0.000000 1.000000
-------------------------------------
```

In this second monte-carlo example, the random variable algebra is illustrated. As previously mentioned, the KSL provides for the ability to define new random variables that are functions of existing random variables. This simple example illustrates how convolution could be used to define and generate binomial random variates.

```
01        // make 10 Bernoulli random variables
02        // and one random variable that is the sum of the 10
03        var binomial: RVariableIfc = BernoulliRV(0.4)
04        for (i in 1..9){
05            binomial = binomial + BernoulliRV(0.4)
06        }
07        val data = binomial.sample(10000)
08        val f = IntegerFrequency("Frequency Tabulation")
09        f.collect(data)
10        println(f)
```

In line 3, an instance of Bernoulli random variable is created. The for-loop starting at line 4 creates 9 additional instances and adds the instances to the starting instance. The "+" operator has been overloaded to return a new instance that is the functional sum of the instances. Then, lines 7-10 sample and collect an integer tabulation of the results. A portion of the results are shown here.

```
Value   Count     Proportion
0    55.0      0.0055
1    389.0     0.0389
2    1201.0    0.1201
3    2189.0    0.2189
4    2425.0    0.2425
5    2028.0    0.2028
6    1129.0    0.1129
7    448.0     0.0448
8    120.0     0.012
9    14.0      0.0014
10   2.0    2.0E-4

Name Frequency Tabulation
Number 10000.0
Average 4.021399999999974
Standard Deviation 1.5585201116744858
Standard Error 0.015585201116744858
Half-width 0.0305501037295127
Confidence Level 0.95
Confidence Interval [3.9908498962704613, 4.051950103729487]
```

Notice that the sample proportion are what you would expect from a Binomial (n=10, p = 0.4) random variable. This same approach can be used to define *arbitrary* functionals of random variables that involve transforms functions such as the natural log, etc. These random variables can then be used in any simulation experiment.

The next example illustrates the process view functionality of the KSL. For examples of the event-view, the reader may refer to Chapter 4 and 5 of Rossetti (2023a). This example is the time-shared computer model from Law (2007).

A time-shared computer system is a type of mainframe computer in which users have access to the CPU of the computer through terminals from which they can submit computer jobs. A user sits at any one of n terminals and submits a job after some "think" time. The thinking time for a user is exponentially distributed with a mean of 25 seconds. No job can enter the CPU unless it is submitted via one of the terminals and there can only be one job submitted at a time from a terminal. In other words, the user must wait for the result of the job before re-submitting from the terminal. Thus, the number of terminals limits the maximum number of jobs in the system. Each job that is submitted requires a random amount of CPU time, which is exponentially distributed with a mean of 0.8 seconds. In a time-sharing system, the CPU is shared across the jobs that currently need executing. In this particular system, each job is executed in a round-robin fashion by holding each job in a queue. The job that currently has the CPU is given a maximum service quantum (allocation), q = 0.1 seconds. If the service time remaining for the job is less than the quantum, then the job will be executed for the remaining amount of time. After its service is entirely completed, it is returned to the terminal. If the service time remaining for a job is greater than the quantum, the job will execute for q = 0.1 seconds and then be swapped back into the queue. The time that it takes to swap the job is fixed and given by $\tau = 0.015$ *seconds. Thus, the total time that the job will use the CPU is* $q + \tau$ *whenever the remaining service time is greater than q = 0.1. After executing for* $q + \tau$*, the job will be placed back at the end of the queue to await its next service allocation. In this system, the key performance measure is the response time for the job. That is, the time from when the job was submitted to when it is returned back to the user. Since the jobs compete for service, the key design question is how many terminals should be provided so that users still receive adequate service (e.g. 30 seconds or less response times).*

One approach for the modeling of this situation via the process view involves modeling two processes: one for the thinking process and the other for the computing process. This type of modeling can be performed withing the KSL by sub-classing from the *ProcessModel* abstract class. This will provide access to defining and using entities and their processes. The basic process definition for the "thinking" process can be implemented by creating an inner class within an instance of a *ProcessModel* that sub-classes from the *Entity* class. The user then must define a coroutine via the *process()* builder function. Line 2 starts the process definition, with lines 3-14 defining the body of the process. The process builder function permits a domain specific language (DSL) for various functions that may suspend the coroutine. Notice that in line 2, that a standard flow control construct of Kotlin is used. Except for lines 5 and 9, this is normal Kotlin code. Line 5 contains a suspending function, *delay(),* that will suspend the process for the indicated time. Line 9 illustrates a very useful suspending function to suspend the current process until another process completes. In this context, the thinking process is suspended until the computing process is completed.

```
01    private inner class Terminal : Entity() {
02        val thinkingProcess: KSLProcess = process() {
03            do {
04                myNumTerminalsThinking.increment()
05                delay(myThinkingTimeRV)
06                myNumTerminalsThinking.decrement()
07                val job = ComputerJob()
08                myNumJobsInProgress.increment()
09                waitFor(job.computingProcess)
10                myNumJobsInProgress.decrement()
11                myResponseTime.value = time - job.startTime
12                numJobsCompleted.value = numJobsCompleted.value + 1
13            } while (numJobsCompleted.value <= myNumJobs)
14        }
15    }
```

The computing process represents what happens to the job while it is using the time-share computer.

```
01        private inner class ComputerJob : Entity() {
02            val startTime = time
03            var remainingServiceTime = myServiceTimeRV.value
04            val runtime: Double
05                get() = if (myQuantum < remainingServiceTime) {
06                    myQuantum + mySwapTime
07                } else {
08                    remainingServiceTime + mySwapTime
09                }
10
11            val computingProcess: KSLProcess = process() {
12                while (remainingServiceTime > 0) {
13                    val a = seize(cpu)
14                    delay(runtime)
15                    release(a)
16                    remainingServiceTime = remainingServiceTime - myQuantum
17                }
18            }
19        }
20    }
21
22    fun main() {
23        val m = Model()
24        TimeSharedComputerPV(m)
25        m.numberOfReplications = 20
26        m.simulate()
27        m.print()
28    }
```

Lines 4-9 illustrate the use of a Kotlin computed property that computes the run time for the job based on its remaining service time. The computing process is defined in lines 11-18. Again, notice that standard programming flow of control (e.g. while-loop) can be used within a process definition. In this example, the process definition uses the commonly used seize(), delay(), and release() suspending functions to have the job receive an allocation from the CPU resource, delay for the running time, and then release the allocation from the resource. Lines 23-27 show how to create a model, specify some run parameters, and execute the simulation. While not presented here, the model of the time-shared computer system was also implemented within a commercial software product and the results from both models were within statistical error.

5 SUMMARY

The KSL provides support for Monte-Carlo simulation, the event view and the process view. Besides support for simulation, the KSL has functionality to capture results into a database, apply data frames to analyze results, write results to CSV files, automates the generation of data to analyze the simulation warmup period, facilitates multiple comparison with the best analysis, and provides the ability to perform bootstrapping. Options are also available for Markov Chain Monte Carlo simulation. Material handling and agent-based modeling constructs are under active development.

REFERENCES

Breslav, A. and E. Elizarov. 2023. "Kotlin Coroutines", https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md, Accessed April 12, 2023.

Dagkakis, G. and C. Heavey. 2016. "A Review of Open Source Discrete Event Simulation Software for Operations Research". *Journal of Simulation*, 10:3, 193-206, DOI: 10.1057/jos.2015.9

Dagkakis, G., I. Papagiannopoulos, and C. Heavey. 2015. "ManPy: An Open-Source Software Tool for Building Discrete Event Simulation Models of Manufacturing Systems". *Softw. Pract. Exper.*, 46: 955– 981. doi: 10.1002/spe.2347.

Joslyn, H. and L. Hecht. 2023. "Dev Most Likely to Learn Go and Rust in 2023, Survey Says". The New Stack, Accessed, April 5th 2023, https://thenewstack.io/developers-most-likely-to-learn-go-and-rust-in-2023-survey-says/

Law, A. 2007. *Simulation Modeling and Analysis*. 4th Edition, McGraw-Hill.

L'Ecuyer, P., R. Simard, and W. D. Kelton. 2002. "An Object-Oriented Random Number Package with Many Long Streams and Substreams". *Operations Research* 50: 1073–75.

Peixoto, T. A., J. Rangel, I. Matias, F. Silva, and E. Tavares. (2017) "Ururau: A Free and Open-Source Discrete Event Simulation Software". *Journal of Simulation*, 11:4, 303-321, DOI: 10.1057/s41273-016-0038-5

Olaitan, O., J. Geraghty, P. Young, G. Dagkakis, C. Heavey, M. Bayer, J. Perrin, and S. Robin. 2014. "Implementing ManPy, a Semantic-Free Open-source Discrete Event Simulation Package, in a Job Shop". *Procedia CIRP*, Volume 25, Pages 253-260.

de Moura, A. L., R. Ierusalimschy. 2004. "Revisiting Coroutines". *ACM Transactions on Programming Languages and Systems*. 31 (2): 1–31.

Rossetti, M. D. 2008. "JSL: An Open-Source Object-Oriented Framework for Discrete-Event Simulation in Java". *International Journal of Simulation and Process Modeling,* Vol. 4, No. 1, pp. 69-87.

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)*. 1st and 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, 2023, https://rossetti.github.io/KSLDocs/-k-s-l-core/ksl.utilities/index.html

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

Rossetti, M. D., B. Aylor, R. Jacoby, A. Prorock, and A. White**.** 2000. "Simfone′: An Object-Oriented Simulation Framework". *Proceedings of the 2000 Winter Simulation Conference,* edited by J. Joines, R. Barton, P. Fishwick, and K. Kang, eds., Piscataway, New Jersey: Institute of Electrical and Electronic Engineers, pp. 1855-1864.

Smith, J. and B. L. Nelson. 2015. "Estimating and Interpreting the Waiting Time for Customers Arriving to a Non-Stationary Queueing System". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz, K.V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, 2610–2621. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Stadler, L., T. Würthinger, and C. Wimmer. 2010. "Efficient Coroutines for the Java Platform". In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java* (PPPJ '10). Association for Computing Machinery, New York, NY, USA, 20–28. https://doi.org/10.1145/1852761.1852765

Team SimPy. 2017. *Simpy*. Retrieved from https://simpy. readthedocs.io/

Weatherly, R. M. and E. H. Page. 2004. "Efficient Process Interaction Simulation in Java: Implementing Co-routines within a Single Java Thread". In *Proceedings of the 2004 Winter Simulation Conference*, edited by R.G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds. 1437–1443, Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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 will be the General Chair for the WSC 2024. He can be contacted at rossetti@uark.edu and https://rossetti.uark.edu/