Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

A TUTORIAL ON RESOURCE MODELING USING 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. The library provides an API framework for developing, executing, and analyzing models using both the event view and the process view modeling perspectives. This paper provides a tutorial on modeling with resources within simulation models. The KSL will be utilized to illustrate important concepts that every simulation modeler should understand within the context of modeling resources within a simulation model. A general discussion of resource modeling concepts is presented. Then, examples are used to illustrate how to put the concepts into practice. While the concepts will be presented within the context of the KSL, the ideas should be important to users of other simulation languages. This tutorial provides both an overview of resource modeling constructs within the KSL and presents tutorial examples.

1 INTRODUCTION

The Kotlin Simulation Library (KSL) is an open-source library for discrete-event system modeling written in the Kotlin programming language. Rossetti (2023a) provides a comprehensive description of the KSL's capabilities, with a summary available in Rossetti (2023d). The KSL extends the Kotlin programming language by offering application programming interfaces and classes for generating pseudo-random numbers, random variate generation, statistical analysis, and building discrete-event system simulation models using both event and process views. The KSL modeling and simulation packages include classes and interfaces that simplify the creation of discrete-event simulation models. The simulation package features the *Model* and *ModelElement* classes, which are the foundational components of KSL models. Additionally, the modeling package includes specialized model elements (sub-classes of *ModelElement*) for modeling common system components like resources, queues, and entities, which are frequently used in simulation modeling. This paper focuses on the resource modeling constructs within the KSL and provides a tutorial for those new to simulation modeling or familiar with other simulation languages concerning important concepts when modeling resources.

Kotlin has consistently ranked within the top 15 programming languages over the past five years, and according to Tran (2025), it currently holds the 13th position, with its popularity steadily rising. Kotlin boasts several appealing features, including clear and concise functional and object-oriented specifications, static compilation, type safety, explicit null representation, and asynchronous programming through co-routines. These capabilities make it a preferred choice over languages like Python and Java. In addition, its compatibility with and similarity to Java make it very appealing to Java programmers. Since Java has long been the first or second ranked programming language, programming in Kotlin can leverage and enhance the Java ecosystem. Finally, building on the long-standing use of co-routines in early simulation languages such as Simula and Simscript II.5 (Kiviat, et al. 1983), the KSL introduces the process view to the Java Virtual Machine ecosystem.

The paper is organized as follows. The next section presents background on resource modeling concepts, primarily from discrete-event simulation modeling literature and books. This coverage is meant to set the stage for exploring resource modeling, in general, and in particular within the KSL. Then, section

3 presents background on the KSL with a focus on the functionality for supporting resource modeling. Some of the technical issues of implementing resource constructs within general purpose programming languages and especially within a coroutine context are touched upon. Section 4 presents a few illustrative examples meant to provide a brief tutorial on resource modeling concepts. The paper ends with a summary and a brief discussion about further issues to consider in resource modeling.

2 BACKGROUND AND LITERATURE REVIEW

The modeling of resources in simulation matters because of the wide variety of contexts within logistics, distribution, manufacturing and healthcare in which simulation is applied. For example, Jun et al. (1999) emphasizes the critical role of resource modeling in healthcare. The authors discuss how discrete-event simulation (DES) can improve efficiency, reduce costs, and enhance patient satisfaction by optimizing resource allocation. The paper highlights the impact of patient scheduling, admissions, routing, and flow schemes on the utilization of resources like physicians, staff, and facilities. The paper underscores the importance of bed, room, and staff sizing and planning to meet patient demand while maintaining high utilization rates. The findings demonstrate that DES effectively models and optimizes resource allocation, improving patient flow, resource utilization, and overall efficiency. This motivates the importance of correctly modeling resources within healthcare delivery systems.

Jenkins and Rice (2007) address the limitations of traditional resource modeling in simulations, which often portray resources as passive entities. The authors propose a more balanced and realistic representation where resources are active and engaging, capturing dynamic interactions and negotiations between resources. The proposed representation is designed to be general, customizable, and reusable across various domains, utilizing the Unified Modeling Language (UML) for its benefits in modeling and documentation. Key entities in the model include Resource, Contract, and Registry, which facilitate dynamic interactions such as negotiation, acceptance, exercise, and termination of contracts.

The proposed model aims to enhance the accuracy and reliability of simulation outcomes by providing a more realistic representation of resources. It emphasizes dynamic interactions, customization, and reusability, making it applicable across different domains such as manufacturing, supply chain management, and military operations. The use of UML promotes standardization and clarity, aiding in the maintenance and updating of models. The authors identify areas for improvement, including scalability, interdisciplinary integration, and adaptability to emerging technologies such as artificial intelligence and the Internet of Things (IoT). They also emphasize the need for user-friendly implementation tools and empirical validation to enhance the practical applicability and relevance of their approach. This motivates simulation modeling practice by highlighting the need for detailed and dynamic resource modeling to improve system performance and decision-making in various industrial applications.

Building on the previously mentioned work, Jenkins and Rice (2009) highlight the limitations of traditional DES models, which often use simplistic representations like "clients" and "servers" waiting in queues, limiting the complexity and realism of the entities. By analyzing over thirty simulation environments, the authors propose a typology for categorizing resource modeling features and suggest a general resource model that may capture the diverse and dynamic nature of real-world resources.

The main contributions of Jenkins and Rice (2009) include a detailed historical analysis of resource modeling approaches, the proposal of a typology categorizing resource modeling features into five levels, and the introduction of a general resource model expressed in the Unified Modeling Language (UML). This model treats clients and servers as equal partners that discover each other through registries and negotiate contracts. The authors also suggest future directions for resource models, advocating for the removal of the distinction between clients and servers to allow for more realistic and complex interactions, similar to agents in multi-agent systems. These suggestions have practical implications for various industries, enhancing the accuracy and applicability of simulation models in representing real-world systems.

Wautelet et al. (2012) address the inadequate representation of resources in traditional engineering methodologies by proposing an ontology centered on resource usage, providing a unified and standardized way to model various types of resources. This ontology emphasizes dynamic resource handling, facilitating

real-time allocation and reservation, crucial for optimizing resource utilization in resource-intensive domains. It supports the development of heterogeneous monitoring systems to enhance interoperability and integration of resources, particularly in industrial contexts. Applied to a case study in the steel industry, the model demonstrates practical applicability and effectiveness in improving structural complexity, resource utilization, and production planning. Despite its strengths, the ontology has shortcomings such as complexity of implementation, limited empirical validation, and scalability concerns. The authors emphasize the need for empirical validation, dynamic models, and educational resources to enhance the model's teaching and learning.

Schriber et al. (2015) provides a comprehensive analysis of how different discrete-event simulation (DES) tools model resources. The authors emphasize the importance of understanding the internal mechanisms of DES software for effective modeling. The paper compares the implementation of entity management rules in four specific DES tools: AutoMod, SLX, ExtendSim, and Simio, highlighting differences in how these tools handle entity states, event scheduling, and resource allocation. Key aspects include the use of Current Event Lists (CEL), Future Event Lists (FEL), and Delay Lists for managing entities, as well as the handling of resource allocation through immediate or deferred methods. The paper provides clear and general states for entities. The paper underscores the practical implications of understanding DES software internals, providing examples of scenarios such as re-capturing resources and handling condition-delayed entities. Overall, the paper provides insights into the implementation details of different DES tools and their practical implications for modelers, ultimately leading to more accurate, reliable, and efficient simulation models.

The following sections will emphasize how the KSL models resources by discussing the internal mechanisms available for handling the special cases mentioned in Schriber et al. (2015).

3 RESOURCE MODELING WITHIN THE KSL

This section provides an overview of the resource modeling capabilities of the KSL. This should set the stage for better understanding the illustrative examples presented in Section 4. While the KSL provides both the event view and the process view for discrete event modeling, the focus here will be on the process view constructs, especially as they relate to the modeling of resources. For details of the event view, the interested reader should refer to Chapter 4 of Rossetti (2023a).

The KSL provides the process view for modeling discrete event dynamic systems by leveraging the coroutine functionality provided in the Kotlin programming language. Coroutines allow for the suspension and resumption of code within a (memory) context to permit an implementation of structured concurrency (Chen and You, 2023). Within a structured concurrency paradigm, a language typically requires some mechanism to indicate which functions utilize coroutines. In Kotlin, these functions are denoted with the keyword, *suspend*. The KSL has an interface, called *KSLProcessBuilder*, that defines 76 suspending functions that facilitate process view modeling. The process view describes processes and their interactions, especially the competition among concurrent processes for some limited resources.

Given the competition between entities executing processes, we define a resource as something that an entity needs or uses to complete its processing (Rossetti, 2015). If the limited availability of the resource restricts the processing of the entity, then the entity suspends its processing. With such a general definition, there may be many contexts that require the modeling of different kinds of resource situations. Many simulation languages have constructs that, in essence, model different kinds of resources, e.g. capacitated resources, buffers, transporters, blockages, conveyors, etc. The KSL has general capabilities, similar in some sense to Simscript II.5, for suspending and resuming processes. These fundamental constructs lead to specialized representations such as hold queues, blocking and signaling, blocking queues, generalized "waitFor", blockages (semaphores/locks), etc.

Like the discussion in Schriber et al. (2015), the focus of this paper is on capacitated resources (e.g. the RESOURCE module of Arena/SIMAN, Resources of AutoMod, Control Variables of SLX, Blocks or Resource Pools of ExtendSim, ResourceObject of Simio, RESOURCE of Simscript II.5, etc.). That is,

using the entity state definitions of Schriber et al. (2015), we are interested in the "condition-delayed state", where the condition is related to the availability or unavailability of a capacitated resource.

To illustrate the basic process view modeling of the KSL, let's consider modeling a simple M/M/1 queuing situation. Suppose customers arrive to a single server according to a Poisson process with rate 1 customer per time unit. The service time is exponentially distributed with a mean of 0.7 time units. The following KSL code models this situation. The code defines a class called *SimpleServiceSystem* to represent the system. The class is parameterized by the number of servers and random variables to represent the time between arrival and service distributions.

```
class SimpleServiceSystem(parent: ModelElement, numServers: Int = 1,
    ad: RandomIfc = ExponentialRV(1.0, 1),
    sd: RandomIfc = ExponentialRV(0.7, 2),
    name: String? = null
) : ProcessModel(parent, name) {
    init {
        require(numServers > 0) { "The number of servers must be >= 1" }
    }

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

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

    private fun arrival(event: KSLEvent<Nothing>) {
        val c = Customer()
        activate(c.serviceProcess)
        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 *SimpleServiceSystem* class subclasses the *ProcessModel* class, which allows the modeling of the process view. The key resource modeling construct is declared via the *ResourceWithQ* class. The *initialize()* function schedules the arrival event for the first customer. The arrival event creates an instance of the *Customer* class and tells the instance to activate (start) its service process. The event then reschedules the next arrival. The magic of coroutines occurs within the *Customer* class. By subclassing from the *Entity* class, the *process()* builder function can be used. The builder function is used to define an instance of the *KSLProcess* class and assign that instance to the *serviceProcess* property. For the purposes of resource modeling, the key suspending function is the *seize()* function. If the entity (process coroutine) that is executing the *serviceProcess* reaches the *seize()* function and the resource is busy, then the coroutine suspends and places the entity in the queue associated with the resource. According to the state definitions of Schriber et al. (2015), the entity will be in the "Condition-Delayed State". Once the resource becomes available, the entity will be removed from the queue and its process resumed.

The *delay()* function is another suspending function that places the entity in the "Time-Delayed State" as per Schriber et al. (2015). The *delay()* function causes the coroutine to suspend until the scheduled end of the delay, at which time the entity's process coroutine is resumed. The *release()* function is also defined within the *KSLProcessBuilder* interface; however, it is not a suspending function. The *release()* function causes the allocation of the resource to the entity to be unallocated and the resource made available for allocation to other competing entities. Notice the "val a = seize(servers)" line. The variable "a" is an instance of an allocation noting that the entity is using the resource. The allocation is then used in the *release()* function to return the allocated units to the resource. With a basic idea of how the KSL represents the process view, we can discuss the basic functionality of the KSL resource modeling constructs.

The resource modeling of the previous example used the *ResourceWithQ* class of the KSL. The *ResourceWithQ* class is a subclass of the more general *Resource* class. In the typology discussed in Jenkins and Rice (2009), these types of resources are passive. That is, they are used by active entities but do not (generally) have process (life) of their own.

An instance or subclass of *Resource* represents a capacitated resource having a set of units that can be allocated to entities. The units of capacity of a resource are all the same. That is, they are indistinguishable with respect to meeting requests from entities. The main difference between the *Resource* class and the *ResourceWithQ* class is that the *ResourceWithQ* class has a prespecified queue for holding requests while the *Resource* class relies on the modeler to specify the queue for waiting requests at the time of requesting units via the *seize()* function. This enables entities to wait in different queues for the same resource.

Since all units of a resource are considered indistinguishable, when an entity first makes a request, the check for whether a resource can fill the request depends upon if the requested number of units are available and the current state of the resource. The KSL allows the capacity of a resource to change with respect to time. Thus, the KSL assumes three states for a resource, busy, idle, and inactive. Define $b(t)$ as the number of units allocated at time t, and $c(t)$ as the current capacity of the resource.

- If $b(t) = 0 \; and \; c(t) = 0$, then the resource is considered inactive.
- If $b(t) > 0 \; and \; c(t) >= 0$, then the resource is busy (but could be active or inactive due to the timing of capacity changes).
- If $b(t) = 0 \; and \; c(t) > 0$, then the resource is idle (and active).

Thus, the states busy, idle and inactive concern the resource as a whole. Provided that the resource is active, the number of available units, $a(t)$, is defined as $a(t) = c(t) - b(t)$.

There are a number of issues to consider when attempting to gain control of a resource's units. The first issue to consider is how to determine which request should be processed first when two requests arrive at exactly the same simulated time to a resource. That is, how should seize requests be handled that occur at the same simulated time? Within a coroutine, the lines of code between suspensions of the coroutine represent instances in time. That is, they correspond to the actions (state changes) associated with events. All events are processed sequentially. Thus, an understanding of the ordering of events becomes essential in understanding this issue.

The natural ordering of events within the KSL executive is first by smallest time, then by smallest priority value, then by smallest order of creation. If two events have the same scheduled time and priority, then whichever event was instantiated first by code execution goes first. Thus, to address the simultaneous arrival of two events, the KSL relies on the modeler to specify a priority when seizing a resource. The seize priority is used to order the events such that the seize with the lower priority value will get the first opportunity to attempt to be allocated resource units. Within the KSL, the entity yields control back to the executive for a zero-time duration to allow the ordering of events that occur at the same simulated time to execute (based on priority), then the entities resume their suspension. Although the KSL does not distinguish between a current and future events list as per Schriber et al. (2015), the essence of the implementation is to manipulate the entities scheduled at the current time via a specification of priorities.

The second issue to consider when modeling capacitated resources is how to allocate units to waiting requests. This processing can be much more complex than first anticipated, especially if the capacity of the resource is allowed to change. To keep the exposition simpler, let's first consider the constant capacity case. The issue at hand is essentially what happens when the resource is released. Releasing the resource returns units for allocation. If there are waiting requests, which requests should receive the released units? As noted in Schriber et al. (2015), simulation languages use different methods for handling this situation. Schriber et al. (2015) illustrate this case with the following example. Suppose two entities are waiting for the resource and suppose that the first entity in the line requires two units of the resource, while the second entity requires one unit of the resource. For simplicity, assume that only one unit of the resource has been released, how should the newly released unit be allocated? As noted in Schriber et al. (2015), there are at least three possibilities:

1. Both continue to wait, and the unit remains unallocated. That is, the entity requiring 1 unit cannot "jump the line" and receive the released unit. Both entities continue to wait because the first entity does not get its two-unit request filled.
2. We allow partial filling. That is, the first unit receives one of its required two, and both entities continue to wait.
3. The second entity "jumps the line" and receives the released unit before the first entity. Since the second entity's request was satisfied, its process is resumed.

The KSL handles this situation with a two-step process. Based on the specified discipline of the queue, the first step selects requests from the queue that are candidates for allocation. The second step processes the candidates in the order listed until no allocations can occur. The first step is governed by a user-suppliable object that implements the *RequestSelectionRuleIfc* interface. This interface extracts the candidates for possible allocation. The KSL's default rule is to select those requests that can be *fully* allocated by the amount available. In the context of Schriber's cases, this is the third possibility. According to Schriber et al. (2015), AutoMod, GPSS/H, and SLX all implement option 3. In the case of the KSL, the user can change this behavior by suppling a different rule for the *queue* to use. For example, the user can implement a rule that allows partial filling of requests.

Since the selection of requests occur from the queue, in the KSL, this behavior is governed by the *RequestQ* class. The user can decide both the queue discipline and how the requests are selected for allocation. The second step of the process is to continue to allocate to the candidates until all the released units are allocated or until there are no waiting requests that are candidates. Thus, if an entity releases more than one unit of a resource, depending on the arrangement of the queue, multiple waiting requests can be resumed at the same simulated time. As in the previously discussed simultaneous seizing of a resource, the order of the resumption events can be specified via the use of a user defined resumption priority. That is, resuming from the conditional delay state is processed through the scheduling executive.

As previously noted, situations that allow the capacity of the resource to change with respect to time complicate the situation. Due to space limitations, we will only briefly mention how the KSL functions when capacity can change. A detailed discussion is available in Chapter 7 of Rossetti (2023a). The KSL provides capacity schedules to allow the capacity to change over durations of time. Then, like how Arena operates, the KSL provides the ability to specify capacity change rules. Two rules are available: wait and

ignore. These rules are only invoked when the capacity is decreased when the resource is busy. The wait rule causes the capacity change to wait until the resource finishes processing the current entity, then the capacity change occurs. The ignore rule starts the duration of the capacity change immediately but allows the busy resource to finish processing before invoking actions associated with the change.

In the case of an increase in capacity, we need to consider how to handle the allocation of the new units. Because entities can wait in many different queues for a resource, the KSL provides a mechanism to register queues for notification of the capacity change. Since the order of notification may be important to the allocation process, the KSL provides the *RequestQueueNotificationRuleIfc* interface. The default queue notification rule is to notify the queues based on the order in which requests were added to the queues. That is, the queue with the oldest waiting request is notified first. Of course, users can develop other rules.

Schriber et al. (2015) also discuss the situation of an entity releasing a resource and then immediately trying to seize the resource (at the same simulated time as the release). Again, Schriber et al. (2015) present three logical options. Option 1 has the release immediately perform the allocation. Thus, the active (releasing) entity is not a contender for allocation because it is not waiting in the queue. Option 2 has the allocation deferred until the releasing entity has entered the condition-delayed state. Option 3 has the allocation occur immediately to the releasing entity without processing the queue. Just like Arena and Extend, the KSL implements Option 1.

The last KSL resource constructs to be discussed is that of resource *pools*. Resource pools represent a collection of other resources that can be used to fulfill requests for units from entities. Simulation languages generally have such constructs to facilitate the sharing of resource units between different activities. The primary challenge associated with managing resource pools is the determination of which resources should be used for the allocation process. This adds an additional layer of complexity to situations like we have previously presented. For example, two entities are waiting for units from a resource pool and suppose that the first entity in the line requires two units from the pool, while the second entity requires one unit from the pool. The pool may have many different capacitated resources, each possibly with differing levels of capacity. For example, suppose the pool consists of three single unit resources (all available), and one resource with capacity 3 with two available units. Clearly, there is sufficient total units available (3 + 2) to meet the needs of both waiting entities. Which resources should be selected for the allocation? Should the 3 individual units be allocated to the waiting entities? Should the resource with 2 units available be used to satisfy the request for two units and a single unit capacity resource handle the second waiting request? Other simulation languages, like Arena, address this situation by allowing the user to specify different resource selection rules.

The KSL separates this decision process into two steps: resource selection and resource allocation. Resource selection rules provide a method or algorithm for selecting resources from a list such that the returned list of resources contains sufficient total amount available to fill the request. If the list is empty, then no allocation can be made. Resource allocation rules provide a method or algorithm for how to allocate the available units across a list of resource that have sufficient available units to meet the request's requirement. The default resource selection rule for the KSL is to select resources in the order in which they were added to the pool until the total available units (across all resources) is greater than or equal to the amount requested. This approach is essentially providing a priority ordering of the resources in the pool. The KSL has the following resource allocation rules:

- Allocate from each resource listed (from the resource selection rule results) in the order in which they are listed until the amount needed is met.
- Randomly permutes the resource selection rule list and then allocate from the permutation until the amount needed is met.
- Sort the resources returned from the resource selection rule from least utilized to most utilized and then allocate in the resulting order until the amount needed is met.
- Sort the resources returned from the resource selection rule from least seized to most seized and then allocate in the resulting order until the amount needed is met.

- Sort the resources returned from the resource selection rule from most available to least available and then allocate in the resulting order until the amount needed is met.

Additional rules can be easily created by providing a function to compare resources. The KSL has additional rules for mobile resources that involve distance and other spatial considerations. The next section presents an example of how to implement resource pooling and the concept of *active* resources.

4 ILLUSTRATIVE EXAMPLES

This section provides two illustrative examples via KSL code. Because of space limitations only code snippets will be provided; however, the full implementations of the example models are available with the KSL GitHub repository (Rossetti, 2025).

4.1 Resource Pooling Example

This first example, based on Chapter 8 of Rossetti (2023a), considers the use of resources to facilitate the movement of entities between workstations. In this system, parts arrive to a testing and repair work center where the parts first have diagnostics performed by 1 of 2 diagnostic workers. The diagnostic activity determines the sequence of test stations that must be visited by the part. After visiting the required testing stations, the part proceeds to repair, where 1 of 3 available repairpersons effect repairs based on the results of the tests. In this situation, the parts are moved between the workstations by any available worker within the testing and repair work center. Figure 1 illustrates this situation in the form of an activity diagram.

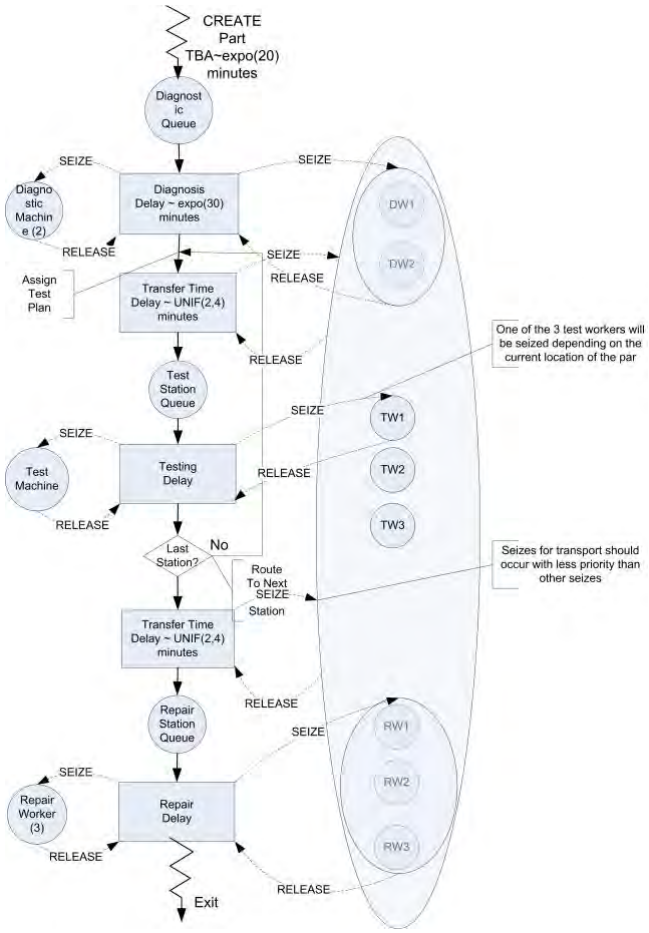

Figure 1: Activity diagram for parts undergoing test and repair.

Notice that in Figure 1, the diagnostics station is staffed by two workers (DW1 and DW2) in the activity diagram. In addition, each testing station has their own worker (TW1, TW2, TW3) assigned. The repair station has three workers (RW1, RW2, RW3). The key item of note is the large ellipse containing all the workers (DW1, DW2, TW1, TW2, TW3, RW1, RW2, RW3). This collection of the workers represents a resource *pool*, for transporting the parts between workstations. The defining characteristic of this situation is that the resources are shared between the workstation activities and the transport activity. In such situations, the modeler needs to decide which activity has precedence (priority) and how to select resources from the pool. The following code will illustrate how to model this situation using the KSL. The first step is to define the resources and the pools.

```
private val dw1 = Resource(this, name = "DiagnosticsWorker1")
private val dw2 = Resource(this, name = "DiagnosticsWorker2")
private val diagnosticWorkers: ResourcePoolWithQ = ResourcePoolWithQ(
    this,listOf(dw1, dw2), name = "DiagnosticWorkersPool")
private val tw1 = ResourceWithQ(this, name = "TestWorker1")
private val tw2 = ResourceWithQ(this, name = "TestWorker2")
private val tw3 = ResourceWithQ(this, name = "TestWorker3")
private val rw1 = Resource(this, name = "RepairWorker1")
private val rw2 = Resource(this, name = "RepairWorker2")
private val rw3 = Resource(this, name = "RepairWorker3")
private val repairWorkers: ResourcePoolWithQ = ResourcePoolWithQ(
    this,listOf(rw1, rw2, rw3), name = "RepairWorkersPool"
)

private val transportWorkers: ResourcePoolWithQ = ResourcePoolWithQ(
    this, listOf(tw1, tw2, tw3, dw1, dw2, rw1, rw2, rw3), name =
"TransportWorkersPool"
)
```

The code uses the previously mentioned *Resource* and *ResourceWithQ* classes to define the individual resources for diagnostics, testing, and repair. Then, via the *ResourcePoolWithQ* class, the defined resources are added to three pools for diagnostics, repair, and transport workers. These pools represent the three ellipses in the activity diagram presented in Figure 1. The following code outlines the process description for this situation. The first two lines of the code illustrate the use of the *EntityGenerator* class. This class is similar to source or create blocks in other simulation languages. The entity generator specifies the time until the first event, the time between events, and the constructor for the type of entity to be created and activated.

Then, the *Part* entity is defined as a subclass of the Entity class. While not shown here, the parts follow a sequence that is randomly selected based on a distribution. This is represented by the *plan* property. Following the same methodology used in the M/M/1 example, the *process()* function is used to describe the coroutine process for the part. The first line of the process increments a response variable to collect statistics on the work-in-process. This is followed by capturing the arrival time within the entity's *timeStamp* property. Since the pattern of seize-delay-release is so common, the KSL provides the *use()* suspending function that combines the three functions into a single function. The *use()* function is called to request a unit of resource from the diagnostic workers pool and then from the transport worker's pool. Notice the use of the *seizePriority* parameter. This causes the request for transport to have pre-defined medium priority. By default, a standard *seize()* call has a higher priority. Thus, the requests for transport workers in the pool will have a lower priority than requests to perform other work in the work center.

Because Kotlin is a general-purpose programming language, all of the constructs of the language are available for use when implementing the process routine. In this example, an iterator to the list of test plans

for the part is used to sequence through the test stations that must be visited. At each step of the iteration the appropriate tester is extracted from the test plan list as well as the time needed for testing. In such approaches, it is easy to implement sequence dependent processing times as part of the testing sequence. Finally, the repair worker pool is used before the number in the system is decremented and the total time in the system captured as output.

```
    private val tba = ExponentialRV(20.0)
    private val myArrivalGenerator = EntityGenerator(::Part, tba, tba)

    private inner class Part : Entity() {
        // determine the test plan
        val plan: List<TestPlanStep> = planList.randomElement

        val testAndRepairProcess: KSLProcess = process(isDefaultProcess = true) {
            wip.increment()
            timeStamp = time
            //every part goes to diagnostics
            use(diagnosticWorkers, delayDuration = diagnosticTime)
            use(transportWorkers, delayDuration = moveTime, seizePriority =
    KSLEvent.MEDIUM_PRIORITY )
            val itr = plan.iterator()
            // iterate through the test plans
            while (itr.hasNext()) {
                val tp = itr.next()
                // visit tester
                use(tp.tester, delayDuration = tp.processTime)
                use(transportWorkers, delayDuration = moveTime, seizePriority =
    KSLEvent.MEDIUM_PRIORITY )
            }
            // visit repair
            use(repairWorkers, delayDuration = repairTimes[plan]!!)
            timeInSystem.value = time - timeStamp
            wip.decrement()
        }
    }
```

As we can see from the code, complex routing and the use of resources can be easily modeled using the KSL. To construct and run the model, the following code can be used.

```
    fun main() {
        val m = Model()
        val tq = TestAndRepairSystem(m, name = "ResourceConstrainedTestAndRepair")
        m.numberOfReplications = 10
        m.lengthOfReplication = 52.0* 5.0*2.0*480.0
        val kslDatabaseObserver = KSLDatabaseObserver(m)
        m.simulate()
        m.print()
        val r = m.simulationReporter
        val out = m.outputDirectory.createPrintWriter("results.md")
```

```
        r.writeHalfWidthSummaryReportAsMarkDown(out, df = MarkDown.D3FORMAT)
    }
```

This code runs the model for ten replications consisting of 52, 5-day weeks, consisting of 2 shifts of 480 minutes. A database is defined to capture the simulation results. In addition, the basic output is captured in the form of a MarkDown table. Table 1 shows a portion of the MarkDown table that was cut and pasted into this document. The output shows that the utilization is reported for the individual resources within a pool. In addition, the number of busy units and fraction of busy units is reported for the pool (across the resources in the pool). Finally, the queueing statistics for the waiting for the resources in the pool are automatically reported.

Table 1: Example output for test and repair example.

Name	Count	Average	Half-Width
DiagnosticsWorker1:InstantaneousUtil	10	0.869	0.004
DiagnosticsWorker1:NumBusyUnits	10	0.869	0.004
DiagnosticsWorker1:ScheduledUtil	10	0.869	0.004
DiagnosticsWorker2:InstantaneousUtil	10	0.808	0.007
DiagnosticsWorker2:NumBusyUnits	10	0.808	0.007
DiagnosticsWorker2:ScheduledUtil	10	0.808	0.007
DiagnosticWorkersPool:NumBusy	10	1.677	0.011
DiagnosticWorkersPool:FractionBusy	10	0.839	0.006
DiagnosticWorkersPool:Q:NumInQ	10	2.326	0.114
DiagnosticWorkersPool:Q:TimeInQ	10	46.457	2.256
TransportWorkersPool:NumBusy	10	7.179	0.038
TransportWorkersPool:FractionBusy	10	0.897	0.005
TransportWorkersPool:Q:NumInQ	10	1.701	0.15
TransportWorkersPool:Q:TimeInQ	10	8.769	0.743

4.2 Active Resource Example

The typology discussed in Jenkins and Rice (2009) pays special attention to distinguishing between passive and active system components. A passive component does not embody "self-generated" decision logic. Instead, passive components are used by entities as the entity interacts with the system. Capacitated resources as discussed in Section 3 and implemented in many simulation languages are primarily passive constructs. In many situations, embedding "intelligence" into the resource facilitates more realistic modeling. For example, the workers in the previous test and repair example, are in fact, human beings, which act and react according to their own decision processes. While workers are often resources, the processing of items by the worker may have complex decision logic initiated by the worker (resource) rather than the entity being processed. We denote these kinds of resources as active resources. As per Jenkins and Rice (2009) the roles of active entity/passive resource, reverses to active resource/passive entity. Complexity as per level 5 of Jenkins and Rice (2009) comes into play when we have both active entities and active resources. In this section, we redo the classic M/M/1 model from an active resource perspective. While simplified, the situation should be illustrative of the kinds of modeling that is necessary when conceptualizing resources as active.

In this implementation, we will have two process descriptions, one for the customers and one for the server. If you are used to the passive resource paradigm, this will seem strange. However, this approach will enable a great deal of modeling flexibility. Especially important is the notion of modeling a resource, such as a server, with its own process. Modeling a resource as an active component of the system (rather than just passively being called) provides great flexibility. This flexibility is useful when approaching some complicated modeling situations. The key to using to two process flows for this modeling is to communicate between the two processes. While there are a wide variety of approaches to implement this situation, the

following presentation will attempt to simplify the implementation, while trying to highlight the important concepts for generalizations.

To coordinate the interaction between the server and the customer, the implementation uses the KSL *HoldQueue* class. Briefly, the *HoldQueue* class builds on the general ability to suspend and resume processes. When an entity enters a hold queue, it suspends until it is told to remove and resume. The following code defines the hold queues and also illustrates how the processes are started.

```
private val generator = EntityGenerator(::Customer, timeBetweenArrivals,
timeBetweenArrivals)
private val customerWaitingQ = HoldQueue(this, "CustomerWaitingQ")
private val customerInServiceQ = HoldQueue(this, "CustomerInServiceQ")
private val serverWaitingQ = HoldQueue(this, "ServerWaitingQ")

private lateinit var server: Server

override fun initialize() {
    server = Server()
    activate(server.serverProcess)
}

override fun replicationEnded() {
    server.isNotShutDown = false
}
```

There are three hold queues defined. The first hold queue, called *customerWaitingQ*, will hold the customer entities that require service. The second hold queue, called *customerInServiceQ*, will hold customers while they experience service. Finally, the last hold queue, called *serverWaitingQ*, will hold the server when it is idle waiting for a customer to arrive.

In a process interaction approach, the concept of shared mutable state is essential. The hold queues are known (globally) to the entire system. Thus, the entities can interact with each other *through* the shared global references to the queues. In addition, the variable *server* is declared at the class level so that it can be referenced by the customer instances. The *initialize()* function shown in the previous code creates a single server and activates its process at time 0.0. The *replicationEnded()* function ensures that the server is shutdown at the end of a replication. Now, consider the customer's process.

```
private inner class Customer() : Entity() {
    val customerProcess: KSLProcess = process(isDefaultProcess = true) {
        wip.increment()
        // signal server of arrival
        server.callServer()
        // wait for service activity to occur
        hold(customerWaitingQ)
        hold(customerInServiceQ)
        timeInSystem.value = time - createTime
        wip.decrement()
        numCustomers.increment()
    }
}
```

The customer's process is activated by the entity generator instance. This process describes what the customer does. Ignoring the statistical collection, the action that the customer performs first is to call the server. This is like "ringing the bell" to indicate the arrival of the customer. The customer then immediately enters the queue to wait for service. According to the terminology of Schriber et al. (2015), the customer has been placed in the *dormant* state. The customer's process is suspended via the *hold()* suspending function.

The following code shows the implementation for calling the server and the server's process. If the server is waiting in the queue for customers to arrive, the server is identified, removed from the queue, and resumed. That is, the server is resumed to the ready state from the dormant state by the arrival of the customer.

```
private inner class Server() : Entity() {
    var isNotShutDown = true

    fun callServer() {
        if (serverWaitingQ.isNotEmpty) {
            val idleServer = serverWaitingQ.peekNext()!!
            serverWaitingQ.removeAndResume(idleServer)
        }
    }

    val serverProcess: KSLProcess = process {
        while (isNotShutDown) {
            hold(serverWaitingQ)
            do {
                val nextCustomer = customerWaitingQ.peekNext()!!
                customerWaitingQ.removeAndResume(nextCustomer)
                myNumBusy.increment()
                delay(serviceTime)
                customerInServiceQ.removeAndResume(nextCustomer)
                myNumBusy.decrement()
            } while (customerWaitingQ.isNotEmpty)
        }
    }
}
```

While the server is not shutdown, the server will continue to serve customers or wait until a customer arrives "to ring the bell". The inner do-while checks if the customer waiting queue is not empty, if it contains waiting customers, the next customer is identified, removed and resumed. This causes the customer to move to the second *hold()* within its process, i.e. the hold queue that contains customers that are in service. Meanwhile, the server starts the delay for service. After completing the delay, the server causes the customer waiting in service to resume. Finally, the customer completes its process by collecting statistics.

In this active resource implementation, it should become apparent that the server is "in charge". The server checks the queue. The server initiates the service activity. The server causes the customer to step through its process. The customer's main action is to tell the server that it has arrived. If we run this model and the model described in Section 3, the statistical results are exactly the same as indicated in Table 2 and 3. In Table 2, the utilization of the server is 0.698. From Table 2, the utilization can be determined from either the *NumBusy* (number of busy servers) or from the *CustomerInServiceQ:NumInQ* response, both indicating 0.698. The statistics for the number in the system and time spent in the system are also the same. The implementations yield exactly the same statistical results because both models are driven by the same random number streams. That is, common random numbers are used. Although no formal testing was

performed, the execution time of the two models was about the same, with the active resource model's execution time slightly shorter. Further investigation of execution efficiency would be an interesting topic for investigation.

Table 2: Results from seize-delay-release passive resource model.

Name	Count	Average	Half-Width
Servers:InstantaneousUtil	30	0.698	0.002
Servers:NumBusyUnits	30	0.698	0.002
Servers:ScheduledUtil	30	0.698	0.002
Servers:Q:NumInQ	30	1.598	0.037
Servers:Q:TimeInQ	30	1.6	0.035
Servers:WIP	30	2.296	0.039
MM1:NumInSystem	30	2.296	0.039
MM1:TimeInSystem	30	2.299	0.037
Servers:SeizeCount	30	14982.033	40.602
MM1:NumServed	30	14981.833	40.64

Table 3: Results from hold queue implementation of active resource model.

Name	Count	Average	Half-Width
ActiveResourceViaHQ:NumInSystem	30	2.296	0.039
ActiveResourceViaHQ:TimeInSystem	30	2.299	0.037
NumBusy	30	0.698	0.002
CustomerWaitingQ:NumInQ	30	1.598	0.037
CustomerWaitingQ:TimeInQ	30	1.6	0.035
CustomerInServiceQ:NumInQ	30	0.698	0.002
CustomerInServiceQ:TimeInQ	30	0.699	0.002
ServerWaitingQ:NumInQ	30	0.302	0.002
ServerWaitingQ:TimeInQ	30	1.001	0.006
ActiveResourceViaHQ:NumServed	30	14981.833	40.64

5 SUMMARY AND CONCLUSIONS

The important difference between the passive resource and active resource implementations involves the structure of the solution. We see from the active resource implementation that complex logic could be readily inserted to select the customer from the queue within the server's process. Secondly, the *callServer()* function offers the opportunity to have complex logic for determining which server should respond to the "bell". To implement a multi-server situation, a collection or pool of active servers could be registered and a general dispatching mechanism implemented to send requests for service to the pool. Conceptually, this form of modeling can be very useful.

For example, consider the modeling of a super center department store with many associates staffing the store. A passive resource approach would "put the customer" in charge, seizing and releasing the workers. Instead, conceptualizing the store with active resources may better facilitate the modeling of the fact that workers perform many kinds of tasks rather than just serve customers. The associate may need to answer the phone, check stockage, take a rest break, staff one part of the store and then move to another, etc. The associate determines what they do next. The active resource perspective should be something that every simulation modeler has in their toolkit.

This paper provides an overview of resource modeling concepts that are found in commercial and open-source simulation packages. It is important for the simulation modeler to understand how their software handles important cases as outlined in Schriber et al. (2015). This paper illustrates how the KSL handles

these classic cases and how the KSL facilitates the incorporation of resource selection and allocation rules and concepts. Finally, the paper compares and contrasts the passive resource and the active resource modeling paradigms and illustrates how active resource models may provide a flexible approach for various modeling situations.

REFERENCES

Banks, J., J. S. Carson, B. L. Nelson, and D. M. Nicol. 2005. *Discrete-Event System Simulation*. 4th ed. Upper Saddle River, New Jersey: Prentice-Hall, Inc.

Jenkins, C. M. and S. V. Rice. 2007. "A General Model of Resources Using the Unified Modeling Language". In *Huntsville Simulation Conference*, HSC 2007, Huntsville, AL, United States.

Jenkins, C. M., and S. V. Rice. 2009. "Resource Modeling in Discrete-Event Simulation Environments: A Fifty-Year Perspective". In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, 755-766 https://doi.org/10.1109/WSC.2009.5429689.

Jun, J. B., S. H. Jacobson, and J. R. Swisher. 1999. "Application of Discrete-Event Simulation in Health Care Clinics: A Survey". *The Journal of the Operational Research Society*, *50*(2) 109–123 https://doi.org/10.2307/3010560.

Kelton, W. D., R. P. Sadowski, and N. B. Swets. 2010a. *Simulation with Arena*. 5th Edition. New York, NY: McGraw-Hill.

Kelton, W. D., J. S. Smith, D. T. Sturrock, A. Verbraeck. 2010b. *Simio & Simulation Modeling, Analysis, Applications*. New York, NY: McGraw-Hill.

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

Rossetti, M. D. 2015. *Simulation Modeling and Arena*. 2nd Edition, Hoboken, NJ: John Wiley & Sons.

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. 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.

Rossetti, M. D. 2025. "KSL Open-Source Repository". Accessed March 5th, 2025, https://github.com/rossetti/KSL.

Schriber, T. J., D. T. Brunner, and J. S. Smith. 2015. "Inside Discrete-Event Simulation Software: How It Works and Why It Matters". In *2015 Winter Simulation Conference (WSC)*, pp. 1-15, https://doi.org/10.1109/WSC.2015.7408149.

Tran, T. 2025. "Top 20 Most Popular Programming Languages in 2024 & Beyond", Access February 23rd, 2025, https://www.orientsoftware.com/blog/most-popular-programming-languages/.

Wautelet, Y., S. Heng, and M. Kolp. 2012. "A Usage-Based Unified Resource Model". In *The 24th International Conference on Software Engineering and Knowledge Engineering*. Redwood City, CA

Chen, Y-A. and Y-P You. 2023. "Structured Concurrency: A Review". In *Workshop Proceedings of the 51st International Conference on Parallel Processing (ICPP Workshops '22)*. Association for Computing Machinery, New York, NY, USA, Article 16, 1–8. https://doi.org/10.1145/3547276.3548519.

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