Proceedings of the 2011 Winter Simulation Conference
S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.

RMSIM: A JAVA LIBRARY FOR SIMULATING REVENUE MANAGEMENT SYSTEMS

Marco Bijvank
Pierre L'Ecuyer
Patrice Marcotte

Department of Computer Science and Operations Research (DIRO)
Université de Montréal
C.P. 6128, Succ. Centre-Ville
Montréal, QC, H3C 3J7, CANADA

ABSTRACT

Revenue management (RM) is the process of understanding and anticipating customer behavior in order to maximize revenue raised from the sale of perishable resources available in limited quantities. While RM systems have been in operation for quite some time, they cannot take into account the full dynamic and stochastic nature of the problem, hence the need to assess them via simulation. In this paper we introduce *RMSim*, a discrete-event and object-oriented Java library designed to simulate large-scale revenue management systems. *RMSim* supports all control policies, arrival processes and customer behavior models hitherto proposed. It can therefore be used to calibrate parameters of the model and to optimize the control policy. A key feature of *RMSim* is that the network RM system can be altered without having to modify the source code of the library. Performance, flexibility and extensibility are the main goals behind the design and implementation of *RMSim*.

1 INTRODUCTION

Revenue management (RM) involves the allocation of scarce resources to stochastic demand for products that consume one or more of these resources, with the aim of maximizing total expected revenue. In application areas such as the airline, hospitality and broadcasting industries, the network configurations and operations can be large and quite complex. For instance, American Airlines serves 250 cities in 40 countries with, on average, more than 3,400 daily flights (American Airlines 2011). Traditionally, mathematical programming methods have been used to determine the control policies, since such methods are well suited to capture network effects. A popular basic technique formulates the RM problem as a static model in which the demand for each product is treated as a deterministic quantity equal to its expected value.

Let us consider a network with M resources and the company sells N products. The incidence matrix $\mathbf{A} = [a_{ij}] \in \{0,1\}^{M\times N}$ defines which resources are used for each product. We let $a_{ij} = 1$ if product j requires a unit of resource i, and $a_{ij} = 0$ otherwise. The total capacity for the resources equals $\mathbf{C} = (C_1,\ldots,C_M)$, the prices are denoted by $\mathbf{p} = (p_1,\ldots,p_N)$ and the average demand for the products equals $E[\mathbf{D}]$. The following deterministic linear program (DLP) is solved to determine the amount of capacity that should be assigned to each product (denoted by $\mathbf{y}$):

$$\begin{aligned} \mathrm{DLP}(\mathbf{C}) = \quad \max \quad & \mathbf{p}^\top \cdot \mathbf{y} \\ \text{s.t.} \quad & \mathbf{A}\cdot\mathbf{y} \le \mathbf{C} \\ & \mathbf{0} \le \mathbf{y} \le E[\mathbf{D}] \end{aligned}$$

Such models oversimplify the complexities of real-world RM systems, and one has to resort to simulation-based models to assess their real-life performance.

978-1-4577-2109-0/11/$26.00 ©2011 IEEE

Discrete-event simulation is well suited at analyzing the performance of complex stochastic systems. Although there exist many discrete-event simulation environments on the market, none (in our knowledge) offers predefined facilities that adequately address the specific issues and features of network RM systems. The purpose of this paper is to introduce a new RM simulation environment (*RMSim*) that can be used to build detailed simulation models for all RM systems currently in use or that have been proposed in the scientific literature. Its primary objective is to analyze the control policies that determine the availability of products to customers. We will also discuss how such analyses can be used to optimize the control policy.

In order to be of practical value, *RMSim* must be able to integrate all aspects of an arbitrary RM system, and to run quickly, even on large networks. Consequently, component reuse and an object-oriented development environment are crucial. This motivated the implementation of *RMSim* as a library of Java classes. In particular, *RMSim* is built on top of the publicly available Java simulation package SSJ (L'Ecuyer and Buist 2005, L'Ecuyer 2008), which provides a powerful simulation engine.

The main challenge of the simulation environment is to achieve flexibility while maintaining efficiency. *RMSim* is not a mere black box that takes model inputs and mysteriously produces model outputs. Different aspects of the revenue management system are implemented in packages that can be used to assemble together a simulation model. Besides its separate components, *RMSim* contains a generic simulator that connects all components of a single RM system in a modular fashion, where a user can specify the network of resources and products, the customer segments including the arrival process and choice behavior per segment, and the type of control policy to accept or reject a customer's request for a certain product (see Figure 1). For each of these ingredients, several predefined possibilities are already offered in *RMSim*, and the users can easily add new ones by adding classes that implement the relevant interfaces. Furthermore, the simulator contains an optimization package based on these components. The user can either specify a data input file according to a fixed schema to perform the optimization and simulation without any programming effort, or assemble (and possibly extend) pieces from the current packages to develop a custom-built simulator tailored to her specific needs. The use of inheritance and well-defined interfaces allows customization of the packages at minimal programming cost.

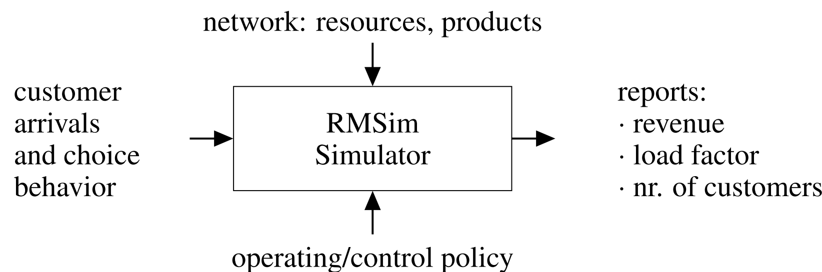

Figure 1: *RMSim* contains separate components to build a simulation model, as well as a pre-constructed generic simulator which assembles components for a single RM system.

The paper is organized as follows. In Section 2, we present an overview of the design principles and general architecture of *RMSim*, as well as a description of its basic components and their interactions. Section 3 provides more details concerning the features and functionalities of the generic simulator implemented in the library. The usage and flexibility of the tool is illustrated by an example in Section 4. In the final Section 5, we hint at future developments. We refer to Bijvank, L'Ecuyer, and Marcotte (2011) for a complete documentation of all classes together with additional examples.

2 GENERAL ARCHITECTURE

The *RMSim* library is composed of packages that cater to the various elements of a revenue management system. Each package is made of independent components that can be combined or extended as required.

Elementary components include information related to resources, products, customers, choice models and the control policy (see Figure 1).

The design principles underlying *RMSim* are to minimize coupling across components and provide high cohesion within components, which provides good flexibility and extensibility. With the use of interfaces and abstract classes we create a loose coupling of the components, where the classes are well encapsulated such that they can be easily extended and re-used. For instance, the abstract class `Controller` provides all basic properties and methods to be used for any control policy, whereas the actual decision on which products to allocate to a customer's request is implemented by a subclass of the `Controller` object. Any new type of control policy that extends the `Controller` class can use the basic methods. Similarly, the interface `CustomerArrivalProcess` specifies which methods need to be implemented and any new arrival process that implements this interface can be used by *RMSim* without the modification of other classes. Relating a class of a subtype to a supertype by the notion of substitutability, such that subroutines or functions of the supertype can be used on elements of the subtype (i.e., subtype polymorphism), makes it possible to use most of the objects without knowing the specific type of the object. The methods and properties of the abstract class and interface are most of the time sufficient. Furthermore, the classes are designed with a well-focused purpose and have only direct access to logically related classes to ensure high cohesion. For instance, the `Controller` object connects the customer and the network (i.e., the products and resources). Based on these design principles, the components interact independently of each other's implementation details. See also Figure 2. Before we discuss the individual components of *RMSim*, we describe this communication mechanism first.

The components interact mostly via an observer design pattern, also known as broadcaster-listener, or publisher-subscriber pattern (see Gamma, Helm, Johnson, and Vlissides 1995). In that framework, one or several objects (the listeners) register their interest in being notified when an event occurs involving another object (the broadcaster). Listeners are implemented as an interface and broadcasters are equipped with a

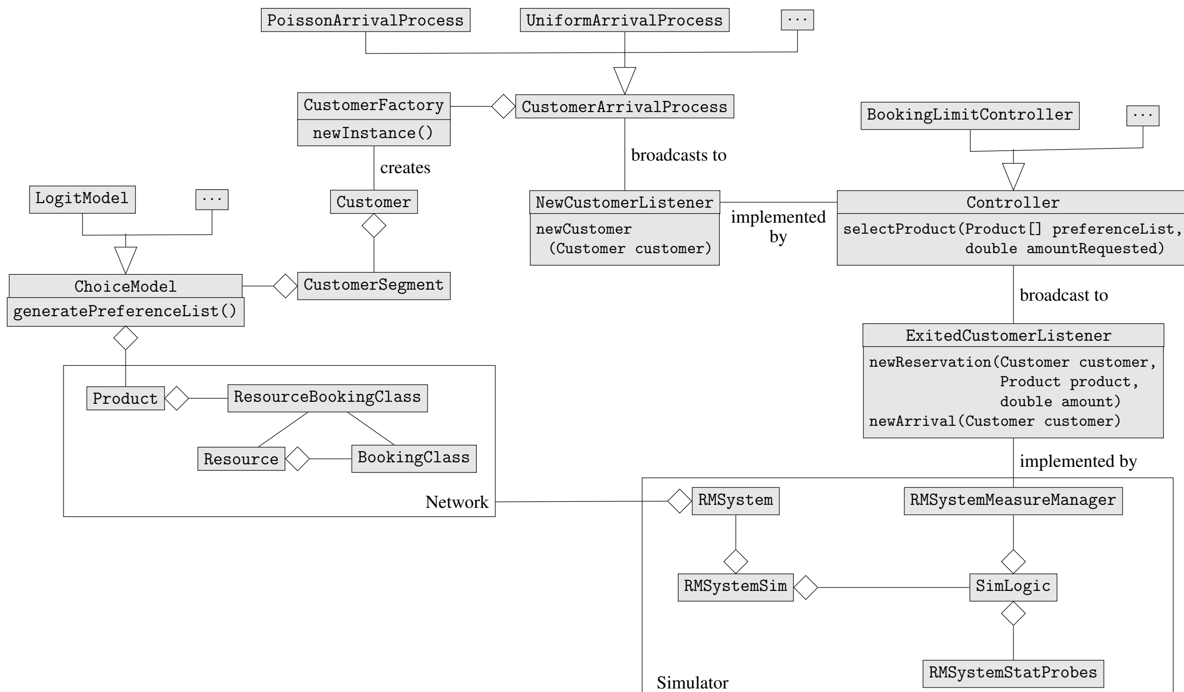

Figure 2: UML diagram of the most important components and their interaction in the *RMSim* library, where a diamond represents a composition relationship and a triangle represents a generalization relationship (i.e., inheritance) (Fowler 2003).

list of registered listeners. For instance, the `Controller` object is a registered listener at any subclass of the `CustomerArrivalProcess`, such that the controller is notified that a decision has to be made regarding a new customer request. The object-oriented observer design pattern ensures that components of this messaging mechanism have limited knowledge about each other to ensure the flexibility to easily extend the packages. We will explain this design pattern more explicitly for the *RMSim* library in this section.

2.1 Simulation Horizon

Since products in a revenue management system are perishable, simulations have a natural finite time horizon T. To cope with the time-varying nature of the arrival process (leisure customers usually book their flight earlier than business customers), we introduce period-change events that specify instants when the parameters of the RM system are modified, and period-change listeners that take care of the appropriate adjustments. Any instance of an object in the library that implements the `PeriodChangeListener` interface can modify specific parameter values over the course of the simulation. For instance, a `Resource` object can control its availability through the modification of bid prices, while `CustomerArrivalProcess` objects may update their arrival rates accordingly.

2.2 Revenue Management Network

A network is represented by M resources and N products. A resource i is characterized by its expiration date $T_i^R \leq T$, its capacity C_i and one or more `BookingClass` objects. Product j is characterized by a set of resources $R(j)$ and booking class combinations $B_i(j)$ for $i \in R(j)$, sold under certain purchase terms and restrictions at a given price. The price p_j of a product j can be specific or set to the sum of the prices for each resource-booking class combination associated with the product. The expiration date of a product j equals $T_j^P = \min\{T_i^R : i \in R(j)\}$. Since prices and capacities can vary dynamically, the objects `Product`, `Resource` and `BookingClass` implement the `PeriodChangeListener` interface, and the first `PeriodChangeEvent` objects are scheduled before the simulation starts. As a result, any network configuration can be managed.

2.3 Customer Segmentation

A customer is represented by an instance of the `Customer` object and is matched to a segment according to a set of attributes, e.g., price, origin-destination, travel time etc. For instance, price-sensitive customers are not willing to pay extra for additional services such as fully or partially refundable products, whereas business customers are willing to spend more money for extra comfort or a specific departure time. Within a segment, customers request the same set of products, but not necessarily in the same preference order. Accordingly, the `CustomerSegment` class specifies the subset of products that customers from this segment may request (the *consideration set*), as well as their number. The generic simulator is able to match customers' preferences and products to generate the consideration set, but the set can also be prespecified.

The choice of a product from the consideration set is prescribed by the abstract class `ChoiceModel`. *RMSim* allows three privileged models of customer choice: exogenous, locational and utility-based, although other models could be considered easily, given the flexibility of the tool. The only requirement imposed is that the model should be implemented as a subclass of `ChoiceModel`, such that the method `generatePreferenceList` has direct access to the products in the consideration set for each individual customer. In the remainder of this subsection, we describe the three types of choice models implemented in *RMSim*.

In the *exogenous model*, a customer has a preferred product i (either deterministic or generated from a distribution). If this product is unavailable, she looks for an alternative product j, based on probabilities α_{ij}. As a result, the choice of the alternative depends on the original choice. In the *locational model*,

products are characterized by a set of attributes, and the preference of an individual customer is specified according to these attributes rather than according to the products themselves. It follows that the initial choice consists in the product that is located closest to its location in the attribute space, and the alternative products are considered in increasing order of their distance from the customer. In the *utility-based model*, a customer assigns a utility U_j to each product in the consideration set, and she selects the product with the highest utility among the set of available products. The utility U_j is usually decomposed in two parts: $U_j = u_j + \xi_j$, where u_j corresponds to a customer's nominal (or expected) utility, and ξ_j, a noisy (random) term. Within this class, we have implemented deterministic models (i.e., u_j is fixed and known, and $\xi_j = 0$) and models where the utility values can vary from one customer to the other according to heterogeneity of preferences among customers (i.e., U_j is modeled as a random variable with a mean equal to the expected utility). Classes of interest include random utility models such as the multinomial logit (MNL) model, where ξ_j follows a Gumbel distribution, or the mixed MNL model where u_j is itself determined according to some distribution. We refer to Train (2009), Shen and Su (2007) and Kök, Fisher, and Vaidyanathan (2009) for further details concerning choice models in RM systems.

2.4 Customer Arrival Process

The stochastic arrival process determines the instants when `Customer` objects need to be created. In most revenue management systems the stream of arriving customers from different customer segments varies over time (Kimms and Müller-Bungart 2007). Arrival processes are generally specified for a finite time period and implement the `PeriodChangeListener` interface. An arrival process has to be specified for each customer segment, such that different types of processes could be used among different customer segments. Currently, *RMSim* embeds the most widely used demand arrival processes in the RM literature, such as the non-stationary Poisson process with arrival rate $\lambda(t)$ over time t. For instance, the rate can be piecewise constant over time, or set to $\lambda(t) = D\beta(t)$ where D is a Gamma distributed random variable and $\beta(t)$ is the density function of a Beta distribution. We also implemented an arrival process where the normal distribution determines the number of arrivals within a fixed time interval, and where these arrivals are uniformly distributed within this interval. Besides these standard arrival processes, others can easily be appended to the *RMSim* library.

In our architectural design, the arrival process does not explicitly specify the segment associated with the customer object, which is created by the arrival process. Rather, we opted for the abstract factory design pattern (Gamma et al. 1995) illustrated in Figure 2. The `CustomerFactory` interface creates new instances of the `Customer` object when the method `newInstance()` is invoked. As a result, the arrival process does not need to know the actual customer segment of the `Customer` object it creates. Instead, a `CustomerFactory` object is passed to the `CustomerArrivalProcess`, which contains all the information to create the appropriate customer's (see also Figure 2). In this framework, it is easy to extend the *RMSim* library to any customized arrival process.

2.5 Controller

Once a `Customer` object is created, the arrival process broadcasts the message that a new customer is created to all registered objects that implement the `NewCustomerListener` interface. A typical object that implements this interface is the `Controller`, since the control policy determines to what extend a customer request is satisfied. A subclass of the `Controller` object implements the actual control policy through the method `selectProducts (preferenceList, amountRequested)`. This interaction between the controller and the arrival process is depicted in Figure 2.

The most common control mechanisms are available in *RMSim*, either quantity-based or price-based. In the latter type, a product is available if its revenue exceeds the sum of the threshold prices (known as bid prices) for all resources associated with this product. That is, a request for product j is accepted if $p_j \geq \sum_{i \in R(j)} \pi(i)$, where $\pi(i)$ is the bid price for resource i associated with product j. Otherwise, the next

product on the customer's preference list (i.e., the ordered consideration set) is considered. Within the class of quantity-based policies, several variants exist. In general, each booking class $b \in B_i$ associated with resource i has a booking limit $q_i(b)$, which represents the amount of capacity reserved for this booking class. The simplest type of control is based on partitioned booking limits, where a customer request for product j is accepted when $q_i(b) > x_i(b)$ for all combinations (i,b) with $i \in R(j)$ and $b = B_i(j)$, where $x_i(b)$ denotes the number of units sold for booking class b on resource i. These booking limits are frequently 'nested' and the booking classes can be reindexed to 'virtual' classes. For a comprehensive account of control policies, we refer to Talluri and van Ryzin (2004). Other control policies can be included in the *RMSim* library through the creation of new subclasses for `Controller`.

Furthermore, since the controller has perfect knowledge of each customer request and the allocated products, the `Controller` object may keep a list of all objects that implement the `ExitedCustomerListener` interface. Based on this feature, the information can be passed to statistical collectors to compute various performance measures, and these are updated as the simulation proceeds.

3 ADDITIONAL FEATURES

As mentioned in Section 1, *RMSim* embeds, besides its basic components, a generic simulator and an optimization package. These are described in this section.

3.1 Generic Simulator

In Section 4, we illustrate by means of an example how the components of the *RMSim* library may be used to build a simulation model. However, the tool also provides a pre-compiled generic simulator that can be used without any programming effort. The input data could be read from data files. Figure 2 provides an overview of relevant objects used by the generic simulator. The constructor `RMSystem()` creates all components of the RM simulation model and links them together, where the counters are declared in `RMSystemMeasureManager` and the statistical collectors in `RMSystemStatProbes`. The three types of performance measures implemented in *RMSim* concern (i) the revenue earned, (ii) the load factor, and (iii) the number of (satisfied) customers. The user of the tool can specify the level of detail for each performance indicator: per product, per resource, per customer segment, per time period or a combination thereof. Besides a textual report on averages, variances and confidence intervals, *RMSim* can produce histograms for graphical reports as well. This feature will also be illustrated in Section 4.

Another feature of the tool is that it can report the actual observations of each individual replication instead of the overall statistics. This has two main advantages: First, it is easy to define new performance measures. Second, individual simulation runs on different revenue management models can be compared, since the generic simulator in *RMSim* exploits the concept of random number streams and substreams, where every random component of the simulation model has its own stream of random numbers. As a result, each random number is used for the same purpose in different configurations. At the beginning of a simulation replication, each random number stream is reset to a new substream to maintain synchronization. This aspect is very useful when we want to compare RM systems that need multiple random number streams, e.g., one for the arrival process and one for making customer's choices in choice models.

To illustrate the performance and scalability of the generic simulator, we consider a network that includes 84 resources with a capacity of 400 units each, and up to 2,000 products in total. In our simulations, we had 16,800 customers arriving to this network. Since each customer's request is processed immediately, the computation time to evaluate the RM system grows linearly with the number of customers. We made an experiment to see how the CPU time increases with the average size of the consideration set for each customer. This set determines the number of products to consider for a customer request. The results have been performed on a 2.0Ghz AMD Opteron 246 processor running Linux and Java, and are presented in Table 1. We see that the average CPU time to simulate this network increases no faster than linearly with

the average size of the consideration set. We also find that our generic simulator can handle thousands of products and customer requests quite rapidly.

Table 1: The CPU time when the average size of the consideration set ranges between 5 and 25 products.

average size of consideration set	5.20	11.08	15.00	19.67	25.64
CPU time (milliseconds)	73	132	186	237	324

3.2 Optimization

The components of the *RMSim* library can also be used to construct an RM model for optimization. The *RMSim* library contains popular optimization techniques such as expected marginal seat revenue (EMSRb), deterministic linear programming (DLP), displacement adjusted virtual nesting (DAVN), and stochastic gradient methods. The latter method is a simulation-based optimization method that uses the generic simulator to generate sample paths during a simulation run. More information on these approaches can be found in Talluri and van Ryzin (2004).

4 EXAMPLE OF A SIMULATOR

In this section, we present a small example to illustrate the features of the *RMSim* library, and the interaction between its components described in Section 2. We consider the same network as studied by van Ryzin and Vulcano (2008), which contains three resources, each endowed with a capacity of 100 units and 2 booking classes (see Figure 3). There are 7 products, either of the high fare (HF) or low fare (LF) type. The price of each product is presented in Table 2. There are 10 customer segments arriving over three booking periods according to a Poisson arrival process, where the average arrival rate and behavioral description are specified in Table 3. The control policy is based on nested booking limits. In this example, we are interested in comparing the performance of two optimization techniques for setting booking limits. For a resource i, we denote the pair of booking limits as $(q_i(\text{LF}), q_i(\text{HF}))$. Using displacement adjusted virtual nesting (DAVN), van Ryzin and Vulcano (2008) obtained the booking limits (71,100), (91,100), and (91,100) for resources AB, AC, and CB, respectively, whereas a stochastic gradient (SG) algorithm gave the booking limits (32,100), (58,100), and (57,100). We want to show how to construct a simulation program with our *RMSim* library to compare the total revenue and compute the load factors of each resource for these two control policies, where the load factor is defined as the percentage of the capacity sold at the end of the simulation horizon T.

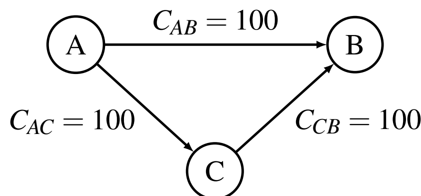

Figure 3: The small network example contains 3 resources and 2 booking classes per resource.

Table 2: The products and their price in the small network example.

product	revenue ($)	product	revenue ($)
HF-AB	300	HF-CB	200
LF-AB	180	LF-CB	100
HF-AC	200	LF-ACB	130
LF-AC	100		

Appendix A contains a Java program that implements this small RM network example. Due to space limitations, we removed the `import` statements. The first part declares the constants that contain the information of Table 2 and Table 3. Next, the variables and methods are declared to create the objects of the RM simulation model. In real-life simulators, the parameters should be read from a data file or a graphical interface as mentioned in Section 3.1. Counters are used to keep track of observations per

Table 3: The customer segments in the small network example.

preference order	description	booking period	arrival rate
LF-AB, LF-ACB	A to B prefer direct	1	60
LF-ACB, LF-AB	A to B price sensitive	1	20
LF-AC	A to C price sensitive	1	10
LF-CB	C to B price sensitive	1	10
LF-AB, HF-AB	A to B buy-up	2	30
LF-AC, HF-AC	A to C buy-up	2	10
LF-CB, HF-CB	C to B buy-up	2	10
HF-AB	A to B only direct flight	3	30
HF-AC	A to C high fare	3	10
HF-CB	C to B high fare	3	10

single simulation replication, whereas statistical probes collecting these observations permit one to compute confidence intervals, histograms, etc., for the performance measures of interest over all replications. In the `main` method at the end of the second part, we first construct the simulator with `new SimpleNetwork()` where the booking limits are based on DAVN. Next, the actual simulation is invoked by `simulate` and the performance measures are displayed in `printStatistics()`. After each replication, the total revenue is stored. Next, based on the same common stream of random numbers, a simulation is performed for a policy where the booking limits are set by the SG procedure. The relative revenue gain of using SG over DAVN is then reported and plotted in a histogram.

The constructor `SimpleNetwork()` creates the objects required to represent the RM network, and connects all components together. For each customer segment, the consideration set and choice model are constructed as well as a customer factory and an arrival process. The piecewise constant Poisson arrival process is registered as a period-change listener to be notified when a new period starts. This allows the arrival rate to be automatically updated over the course of the simulation. These time instances are initiated by the `PeriodChangeEvent` object of each segment. Arrival times are generated with dedicated random variate generators for each customer segment, where the underlying random number stream is constructed by `new MRG32k3a()`. Finally, the controller is constructed and registered as a new-customer listener for each arrival process. A `MyPerformanceMeasure` object is connected to the controller as exited-customer listener for statistical collections. Note how products, resources and booking classes are interconnected, as well as the products in the consideration set of a customer segment, the arrival process and the controller.

Once the RM simulation model is constructed, the method `simulate` performs n independent simulation runs by invoking `simulateOneReplication()` n times. At the start of the initial replication, the statistical collectors are cleared. During each simulation replication, the RM simulation model is initialized, the simulation is performed, and observations are collected. A replication starts with the initialization of the simulation clock, each random number stream is reset to a new substream with `resetNextSubstream()` and all statistical counters are reset to zero. The elements of the revenue management system are initialized to eliminate any side effects of previous replications, and the arrival processes are initialized by invoking the `init()` methods. Next, the first customer arrival of each customer segment is scheduled with the `start()` method of each period-change event. The actual execution of events is initiated with `sim.start()`.

When an *arrival process* triggers a customer arrival, the `newInstance()` method in `Customer-Factory` is called corresponding to the arrival process of the customer segment, and the factory constructs a `Customer` object of the appropriate segment. The method `generatePreferenceList()` is then invoked to order the products in the consideration set for the new customer according to the choice model of the customer's segment. The arrival process broadcasts the new customer to all registered objects

that implement the new-customer listener interface (e.g., the controller) and the next customer arrival is scheduled.

Based on the availability of the product in the preference list and the nested booking limits, the *controller* selects the products to be assigned to the customer's request by calling the method `selectProducts`. When a different subclass of `Controller` is used, a different type of control policy can be specified. Next, the customer and the assigned products (if any) are notified to the registered exited-customer listeners at the controller. As a result, all counters are updated. When customers stop arriving, the simulation replication ends and the observations of the counters are appended to the statistical collectors. Once all replications have been performed, the statistical probes are used to report on the performance measures, and the total revenue of each replication is stored in `revenueDAVN`.

The method `resetForNewSetting()` reinitializes the stream of random numbers to their initial state with `resetStartStream()`, to make sure that the simulation replications with the configuration based on SG use exactly the same sequence of uniform random numbers as the first configuration based on DAVN. Since the parameters of the arrival process are the same for both configurations, this implies that the sequences of customer arrival times will be exactly the same as compared to the simulation runs for the RM system with booking limits based on DAVN (see also Section 3.1). The booking limit of each resource-booking class combination is reset to a value based on SG. The simulation is performed for this setting and the total revenue of each replication is stored in `revenueSG`. Next, we can compute the relative revenue gain of using the booking limits based on SG compared to the values based on DAVN for each replication r by $(\text{rev}_r^{\text{SG}} - \text{rev}_r^{\text{DAVN}})/\text{rev}_r^{\text{DAVN}}$, where rev_r^{SG} and $\text{rev}_r^{\text{DAVN}}$ denote the observed revenue in the r-th replication when the booking limits are based on SG and DAVN, respectively. This is possible, since the two simulations have used the same streams of random numbers. The statistics of this comparison are reported in a histogram, which is written to the file `SimpleNetwork_chart.tex`. This histogram is plotted in Figure 4, whereas the statistical results are presented in Table 4. These results are consistent with those of van Ryzin and Vulcano (2008). Figure 4 also shows histograms, produced by *RMSim* as well, of the load factor for each resource AB, AC, and CB. These histograms give more detailed insights in the different performances of the two sets of bookings limits.

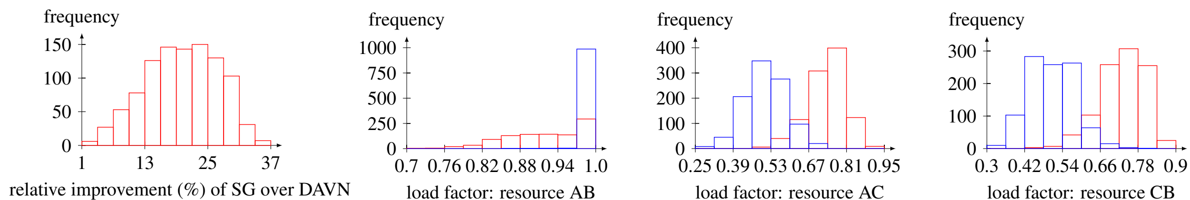

Figure 4: The histogram of the relative revenue gain when a SG algorithm is used to determine booking limits over DAVN for 1,000 simulation replications of the small RM network example, as well as the load factor for every resource (where blue represents DAVN and red SG).

5 CONCLUSION

In this work, we have described *RMSim*, an object-oriented Java-based library for building and analyzing discrete-event simulation models of revenue management systems. It has been designed to be extensible and flexible, while being powerful enough to simulate with ease network systems consisting of thousands of products and customers. As a result, it can efficiently handle large and complex RM models. The hierarchical approach adopted in *RMSim* allows users to tailor the tool to their specific needs, with minimal programming effort. The next step in the design of *RMSim* will involve the consideration of competition between firms having their own RM system, as well as the development of new optimization methods.

Table 4: The statistical results on the performance measures of interest in the small RM network example.

setting	perform. measure	min	max	average	standard dev.	95% conf. interval
DAVN	load factor AB	0.87	1.00	0.999	9.6E-3	[0.998; 0.999]
	load factor AC	0.26	0.73	0.505	0.076	[0.500; 0.510]
	load factor CB	0.32	0.80	0.505	0.074	[0.501; 0.510]
	total revenue	28540	36270	32124	1292	[32043; 32204]
SG	load factor AB	0.70	1.00	0.917	0.066	[0.913; 0.921]
	load factor AC	0.48	0.94	0.733	0.070	[0.729; 0.738]
	load factor CB	0.46	0.90	0.733	0.069	[0.728; 0.737]
	total revenue	30590	45170	38583	2676	[38417; 38750]
comparison	rel. revenue gain	2.35%	35.50%	20.11%	6.90%	[19.68%; 20.54%]

A JAVA PROGRAM OF A SMALL RM NETWORK

```
public class SimpleNetwork {
   static final int M = 3;  // number of resources
   static final int N = 7;  // number of products
   static final int K = 10; // number of customer segments
   static final String[] RESOURCE_NAMES = { "AB", "AC", "CB"};
   static final int[] RESOURCE_EXP = {4, 4, 4};       // the expiration date
   static final int[] RESOURCE_C = {100, 100, 100}; // the initial capacity
   static final String[][] BOOKINGCLASS_NAMES = {{"HF", "LF"}, {"HF", "LF"}, {"HF", "LF"}};
   static final int[][] BOOKINGCLASS_Q_DAVN = { {100, 71}, {100, 91}, {100, 91} }; // the booking limits
   static final int[][] BOOKINGCLASS_Q_SG = { {100, 32}, {100, 58}, {100, 57} }; // the booking limits
   static final String[] PRODUCT_NAMES = {"HF-AB","LF-AB","HF-AC","LF-AC","HF-CB","LF-CB","LF-ACB"};
   static final double[] PRODUCT_P = { 300, 180, 200, 100, 200, 100, 130 };   // the prices
   static final String[][] PRODUCT_R = {{"AB"},{"AB"},{"AC"},{"AC"},{"CB"},{"CB"},{"AC","CB"}};
   static final String[][] PRODUCT_B = {{"HF"},{"LF"},{"HF"},{"LF"},{"HF"},{"LF"},{"LF","LF"}};
   static final double[] SEGMENT_START = {1, 1, 1, 1, 2, 2, 2, 3, 3, 3}; // the starting time of each customer segment
   static final double[][] SEGMENT_DURATIONS = { {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1} }; // period length of the arrival process per segment
   static final double[][] LAMBDAS = { {60}, {20}, {10}, {10}, {30}, {10}, {10}, {30}, {10}, {10} }; // average arrival rate for each customer segment
   static final String[][] PREFERENCESETS = { {"LF-AB", "LF-ACB"}, {"LF-ACB", "LF-AB"}, {"LF-AC"}, {"LF-CB"}, {"LF-AB", "HF-AB"}, {"LF-AC", "HF-AC"},
                                               {"LF-CB", "HF-CB"}, {"HF-AB"}, {"HF-AC"}, {"HF-CB"} }; // consideration set per segment in decreasing order
   static final int NRREPLICATIONS = 1000;
   static final double LEVEL = 0.95;

   // revenue management system components
   Controller controller;                                  // control policy to determine product availability
   Resource[] resource = new Resource[M];                  // resources in the network
   Product[] product = new Product[N];                     // products in the network
   CustomerSegment[] customerSegment = new CustomerSegment[K]; // customer segments arriving to the network
   PeriodChangeEvent[] pce = new PeriodChangeEvent[K];         // period-change event for each arrival process
   PiecewiseConstantPoissonArrivalProcess[] arrivProc = new PiecewiseConstantPoissonArrivalProcess[K];

   // the simulator and its counters and statistical collectors
   Simulator sim = new Simulator();
   double sumRevenue;            TallyStore revenue = new TallyStore ("Total revenue of system");           Tally[] loadFactor = new Tally[M];

   SimpleNetwork() {
      for (int i = 0; i < M; i++) resource[i] = readResource (i);
      for (int j = 0; j < N; j++) product[j] = readProduct (j);
      for (int k = 0; k < K; k++) {
         Product[] considerationSet = readConsiderationSet (k);
         ChoiceModel choiceModel = new DeterministicModel (considerationSet);
         customerSegment[k] = new CustomerSegment (choiceModel);
         pce[k] = new PeriodChangeEvent (sim, SEGMENT_START[k], SEGMENT_DURATIONS[k]);
         arrivProc[k] = new PiecewiseConstantPoissonArrivalProcess (pce[k], new CustomerFactory (sim, customerSegment[k]), LAMBDAS[k], new MRG32k3a());
      }
      controller = new TheftNestedBookingLimitController();
      controller.addExitedCustomerListener (new MyPerformanceMeasure());
      for (int k = 0; k < K; k++) arrivProc[k].addNewCustomerListener (controller);
      for (int i = 0; i < M; i++) loadFactor[i] = new Tally ("Load factor of resource " + resource[i].getName());
   }

   // Creates a resource object based on the input date
   public Resource readResource (int r) {
      final int B = BOOKINGCLASS_NAMES[r].length; // number of booking classes
      BookingClass[] bookingClass = new BookingClass[B];
      for (int b = 0; b < B; b++) bookingClass[b] = new BookingClass (BOOKINGCLASS_NAMES[r][b], BOOKINGCLASS_Q_DAVN[r][b]);
      return new Resource (RESOURCE_NAMES[r], new DateObject (RESOURCE_EXP[r]), RESOURCE_C[r], bookingClass);
   }

   // Creates a product object based on the input date
   public Product readProduct (int j) {
      final int R = PRODUCT_R[j].length; // number of resources
      ResourceBookingClass[] resourceBookingClass = new ResourceBookingClass[R];
      for (int i = 0; i < R; i++) resourceBookingClass[i] = getResourceBookingClass (PRODUCT_R[j][i], PRODUCT_B[j][i]);
      return new Product (PRODUCT_NAMES[j], resourceBookingClass, PRODUCT_P[j]);
   }
```

```
   // Combines the resource and booking class objects based on their names
   public ResourceBookingClass getResourceBookingClass (String rName, String bcName) {
      for (int i = 0; i < M; i++)
         if (rName.compareTo (resource[i].getName()) == 0) return new ResourceBookingClass (resource[i], resource[i].getBookingClass (bcName));
      return null;
   }

   // Selects the products in the choice set for the customers of segment k
   public Product[] readConsiderationSet (int k) {
      final int sizeChoiceSet = PREFERENCESETS[k].length;            Product[] choiceSet = new Product[sizeChoiceSet];
      for (int p = 0; p < sizeChoiceSet; p++) choiceSet[p] = getProduct (PREFERENCESETS[k][p]);
      return choiceSet;
   }

   // Returns the product object based on the name
   public Product getProduct (String pName) {
      for (int j = 0; j < N; j++) if (pName.compareTo (product[j].getName()) == 0) return product[j];
      return null;
   }

   // Sets the booking limits for each resource based on the constant BOOKINGCLASS_Q_SG
   public void resetForNewSetting() {
      for (int k = 0; k < K; k++) arrivProc[k].getStream().resetStartStream(); // reset random streams
      for (int i = 0; i < M; i++) // set the booking limits based on SG for each resource
          for (int b = 0; b < BOOKINGCLASS_NAMES[i].length; b++)
             resource[i].getBookingClass (BOOKINGCLASS_NAMES[i][b]).setBookingLimit (BOOKINGCLASS_Q_SG[i][b]);
   }

   // Updates counters when a customer enters/exits
   class MyPerformanceMeasure implements ExitedCustomerListener {
      public void newReservation (Customer c, Product p, double amount) { sumRevenue += p.getPrice(); }
      public void newArrival (Customer c) {}
   }

   public void simulateOneReplication() {
      for (int k = 0; k < K; k++) arrivProc[k].getStream().resetNextSubstream(); // reset random substreams
      sim.init();                   sumRevenue = 0;     // initialize the simulation and its counters
      for (int i = 0; i < M; i++) resource[i].init(); // initialize the network and the arrival processes
      for (int k = 0; k < K; k++) {             arrivProc[k].init();            pce[k].init();            pce[k].start(); }
      sim.start();                // perform the simulation
      revenue.add (sumRevenue); // update the statistical collectors
      for (int i = 0; i < M; i++) loadFactor[i].add (1 - resource[i].getAvailableCapacity() / resource[i].getTotalCapacity());
   }

   void simulate (int n) {
      revenue.init();               for (Tally t : loadFactor) t.init(); // clear the statistical collectors to be empty
      for (int r = 0; r < n; r++) simulateOneReplication();
   }

   public void printStatistics() {
      System.out.println (revenue.reportAndCIStudent (LEVEL, 3));        for (Tally t : loadFactor) System.out.println (t.reportAndCIStudent (LEVEL, 3));
   }

   public static void main (String[] args) {
      final SimpleNetwork s = new SimpleNetwork();
      s.simulate (NRREPLICATIONS);          s.printStatistics();          double[] revenueDAVN = s.revenue.getArray().clone();
      s.resetForNewSetting();
      s.simulate (NRREPLICATIONS);          s.printStatistics();          double[] revenueSG = s.revenue.getArray();
      TallyStore improvedRevenue = new TallyStore ("Improved revenue");
      for (int r = 0; r < NRREPLICATIONS; r++) improvedRevenue.add (100*(revenueSG[r]-revenueDAVN[r])/revenueDAVN[r]);
      System.out.println (improvedRevenue.reportAndCIStudent (LEVEL, 3));
      HistogramChart hist = new HistogramChart ("", "relative improvement (%)", "frequency", improvedRevenue.getArray(), improvedRevenue.numberObs());
      (hist.getSeriesCollection()).setBins (0,12,1,37);                   hist.setManualRange(new double[]{1,37,0,150});
      hist.getXAxis().setLabels (new double[]{1,7,13,19,25,31,37});      String histLatex = hist.toLatex (6.0, 4.0);
      Writer file = new FileWriter ("SimpleNetwork_chart.tex");          file.write (histLatex);                  file.close();
   }
}
```

Figure 5: The Java program of the small RM network example discussed in Section 4.

REFERENCES

American Airlines, 2011. Accessed March 3, 2011. http://www.aa.com.

Bijvank, M., P. L'Ecuyer, and P. Marcotte. 2011. *RMSim: A Java Library for Simulating Revenue Management Systems*. Département d'Informatique et de Recherche Opérationnelle, Université de Montréal. Software user's guide, forthcoming.

Fowler, M. 2003. *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley, Reading, Massachusetts.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.

Kimms, A., and M. Müller-Bungart. 2007. "Simulation of stochastic demand data streams for network revenue management problems". *OR Spectrum* 29:5–20.

Kök, A., M. Fisher, and R. Vaidyanathan. 2009. "Assortment Planning: Review of Literature and Industry Practice". In *Retail Supply Chain Management*, edited by N. Agrawal and S. Smith, Volume 6, 1–55. Springer, New York.

L'Ecuyer, P. 2008. *SSJ: A Java Library for Stochastic Simulation*. Département d'Informatique et de Recherche Opérationnelle, Université de Montréal. Software user's guide, available at http://www.iro.umontreal.ca/~lecuyer.

L'Ecuyer, P., and E. Buist. 2005, December. "Simulation in Java with SSJ". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 611–620. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Shen, Z.-J., and X. Su. 2007. "Customer Behavior Modeling in Revenue Management and Auctions: A review and New Research Opportunities". *Production and Operations Management* 16 (6): 713–728.

Talluri, K., and G. van Ryzin. 2004. *The theory and practice of revenue management*. Kluwer Academic Publishers, Norwell, Massachusetts.

Train, K. 2009. *Discrete choice methods with simulation*. Second ed. Cambridge University Press, Cambridge, Massachusetts.

van Ryzin, G., and G. Vulcano. 2008. "Computing virtual nesting controls for network revenue management under customer choice behavior". *Manufacturing & Service Operations Management* 10:448–467.

AUTHOR BIOGRAPHIES

MARCO BIJVANK is a post-doctoral researcher in the Département d'Informatique et de Recherche Opérationnelle at the Université de Montréal, Canada. He received a Masters in Business Mathematics and Informatics and a Ph.D. in Exact Sciences from the VU University Amsterdam, The Netherlands. His research interests are in revenue management, inventory theory, operational decision making in logistics processes, and the application of operations research techniques in practice. His email address is bijvankm@iro.umontreal.ca.

PIERRE L'ECUYER is Professor in the Département d'Informatique et de Recherche Opérationnelle, at the Université de Montréal, Canada. He holds the Canada Research Chair in Stochastic Simulation and Optimization. He is a member of the CIRRELT and GERAD research centers. His main research interests are random number generation, quasi-Monte Carlo methods, efficiency improvement via variance reduction, sensitivity analysis and optimization of discrete-event stochastic systems, and discrete-event simulation in general. He is currently Editor-in-Chief for *ACM Transactions on Modeling and Computer Simulation*, and Associate/Area Editor for *ACM Transactions on Mathematical Software*, *Statistics and Computing*, *International Transactions in Operational Research*, and *Cryptography and Communications*. He obtained the *E. W. R. Steacie* fellowship in 1995-97, a *Killam* fellowship in 2001-03, and became an INFORMS Fellow in 2006. More information and his recent research articles are available on-line from his web page: http://www.iro.umontreal.ca/∼lecuyer.

PATRICE MARCOTTE is Professor and Chairman of the Département d'Informatique et de Recherche Opérationnelle, at the Université de Montréal, Canada. He has published more than eighty papers in international journals and currently sits on the editorial board of *Operations Research*, *Transportation Science*, *JOTA* and *Operations Research Letters*, the latter as area editor for continuous optimization. His research interests in traffic models have led him to the study of network equilibrium and, more theoretically, variational inequalities and bilevel programs. Together with a student and colleagues, he is shareholder of a firm that specializes in the development of revenue management software.