Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

A JAVA LIBRARY FOR SIMULATING CONTACT CENTERS

Eric Buist
Pierre L'Ecuyer

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, C.P. 6128, Succ. Centre-Ville
Montréal (Québec), H3C 3J7, CANADA

ABSTRACT

ContactCenters is a Java library for writing contact center simulators. It supports multi-skill and blend contact centers with complex and arbitrary routing, dialing policies, and arrival processes. The programmer can alter the simulation logic in many ways, without modifying the source code of the library. A simulator can interoperate with other libraries, e.g., for optimization and statistical analysis. Performance, flexibility, and extensibility are the main goals of its design and implementation. In this paper, we present the general architecture of the library, its main components and their interaction. We give an example of a contact center simulator and provide comparisons with a widely used commercial simulation system also offering facilities for contact center simulation.

1 INTRODUCTION

A *contact center* is a set of resources (communication equipment, employees, computers, etc.) providing an interface between customers and a business (Mehrotra and Fama 2003, Gans, Koole, and Mandelbaum 2003). A *contact* represents a customer's request for some service such as information, subscription, order, etc. Customers may use various media for contacting a business: telephone, fax, mail, or Internet.

Inbound contacts are initiated by customers trying to communicate with the business. A customer can be *blocked*, i.e., receive a busy signal, if all phone lines are used at the time he calls. He can also be queued if service cannot be started immediately. A queued customer may become impatient and abandon without receiving service.

Outbound contacts are initiated by agents contacting customers, or by a *predictive dialer* making phone calls by trying to anticipate the number of free agents at the time contacted customers are reached. A *right party connect* occurs when an outbound contact is successful. A *mismatch* represents a successful contact that cannot be served immediately.

Modern contact centers use *skill-based* routing for processing different types of requests when each agent is trained for handling only a subset of these types. Each contact is assigned a type (or skill). Before reaching an agent, a customer must indicate his needs: callers interact with an *interactive voice response* (IVR) unit while Internet users enter data in a Web form. Outbound contacts can also have a type, since all customers are not contacted for the same reason.

The agents are partitioned in agent groups or skill sets. All agents in a group share the same skills, i.e., they can serve the same types of contacts (although some members may be more efficient than others).

Queueing theory can be used to derivate approximations for estimating the performance measures of contact centers, but only for models that oversimplify the complexities of real-life systems for which only simulation can provide accurate results.

A contact center can of course be modeled using generic simulation tools, but that could be a very large programming task for complex models. Specialized software, such as Rockwell's *Arena Contact Center Edition* (Bapat 2003), or NovaSim's *ccProphet* (NovaSim 2003) can ease modeling significantly. Supporting multi-skill contact centers with complex routing policies, these point-and-click tools provide convenient graphical user interfaces (GUIs). Many common performance measures can be estimated, and animations can help debugging models. However, the great number of software layers reduces performance, and modeling some aspects not supported by the tool is often difficult, complicated, and can lead to inefficient code.

The *ContactCenters* software tool introduced here provides facilities to construct an event-driven simulator for an almost arbitrary contact center. It is implemented as a package of Java classes, built on top of the SSJ Java simulation library (L'Ecuyer, Meliani, and Vaucher 2002, L'Ecuyer and Buist 2005, L'Ecuyer 2004), which provides

a fast and robust simulation engine. It provides basic building blocks which can be combined and extended to model various systems with great detail. Generic simulators with a lot of flexibility can be constructed with the package and specific ones are already available. Each contact, represented by an entity (i.e., an object) with several attributes, is simulated individually for maximal flexibility. Its path in the system can be arbitrarily complex: waiting in several queues, getting served by several agents, etc.

The Java base offers the advantages of a general-purpose, widely-used, and well-supported programming language. The package can be extended easily to support additional logic and scenarios. By using inheritance, this extension can usually be achieved without complete rewriting of new components. Thanks to the optimizations of modern Java Virtual Machines, simulators written with *ContactCenters* run several times faster than if implemented with the leading commercial simulation software products (see Section 4).

Contact centers are often simulated with the aim of optimizing some performance measures under a set of constraints; e.g., to find a least-cost staffing or routing under quality-of-service constraints (Atlason, Epelman, and Henderson 2004, Cezik and L'Ecuyer 2004). This typically requires a large number of simulation runs, so CPU times are important in that context. While some commercial simulation tools provide an optimizer whose logic is typically hidden, special purpose optimizers adapted to call centers are usually more efficient. They can easily be accessed (or programmed) in Java. Using Java also gives easy access to external libraries such as GUI building tools, statistical and optimization software, etc. The flexibility offered by the library also facilitates the implementation of variance reduction techniques and gradient (or subgradient) estimators (Atlason, Epelman, and Henderson 2003).

The next section of this paper provides an overview of the architecture of *ContactCenters* by presenting the elementary components and briefly summarizing how they interact. Section 3 contains a commented example of a contact center simulator. It illustrates how to use the library. Section 4 compares the performance with that of a commercial simulation tool. Section 5 gives indications on planned future work. A complete documentation of all classes, and several additional examples, are given by Buist and L'Ecuyer (2005) and Buist (2005).

2 GENERAL ARCHITECTURE

The main goal of the *ContactCenters* library is to provide a flexible, extensible, and powerful framework for simulating a large variety of contact centers. Flexibility is obtained by constructing small independent components that can be combined as needed, and extended by using inheritance.

Elementary components include contacts, contact sources, waiting queues, and agent groups. These components can interact without requiring much information about each other via the *observers* design pattern (Gamma, Helm, Johnson, and Vlissides 1998). In that setting, an *observable* object, also called *broadcaster*, can broadcast information to a list of registered listeners known at runtime only. A *listener*, also called an *observer*, is an object receiving information from these broadcasters. In Java, listeners are required to implement a particular interface used by the broadcaster to transmit information through specified methods. In our implementation, each component of the library defines its own listener interface to avoid the necessity of type casting by the observers.

2.1 The Simulation Periods

In contact center models, time is usually divided into periods of 15 to 60 minutes, between which model parameters such as the number of agents in each group, arrival rates of calls, etc., may change. Sometimes, data must be collected separately for the different periods. In the simulation, *period-change events* take care of triggering the appropriate changes. In our implementation, this is done via the observer pattern: at the beginning of each period, an event notifies all registered period-change *listeners*, who in turn make the appropriate adjustments to the model parameters, statistical collectors, etc., under their control. Both fixed-sized and variable-sized periods are supported.

In *ContactCenters*, we assume that the center is opened during P periods, called the *main periods*. Main period p, for $p = 1, \ldots, P$, corresponds to a time interval $[t_{p-1}, t_p)$, where $0 \le t_0 < \cdots < t_P$. Since the simulation must often start before the center opens and stop after it closes (e.g., incoming calls could start getting queued before the center opens and calls in process or in the queue at closing time must be completed), we add two extra periods: The *preliminary period* $[0, t_0)$, during which the center is not yet open, and the *wrap-up period* $[t_P, t_{P+1}]$, which goes from the time the center closes to the time t_{P+1} at which the simulation is over.

2.2 The Contacts

A contact is represented by a data object of class `Contact`, with fields corresponding to attributes such as arrival time, type identifier, priority, etc. At any time, the simulator can access these attributes and modify most of them. A contact entity corresponds to a single customer, but a customer may need to make several contacts before leaving the system.

A contact can be associated with a *trunk group*, i.e., a bank of communication channels such as phone lines. A channel is like a resource which is allocated for the time the contact is in the system. Contacts arriving when

no communication channel is available are blocked. By default, no trunk group is associated with contacts, avoiding capacity limitation.

One is free to define any desired custom attribute by creating a subclass of `Contact` and adding new fields. This permits, e.g., to associate costs or random numbers with contacts. For instance, generating all the required random variates at the construction of a contact, even when some are unused, may be useful for random number synchronization for variance reduction.

2.3 The Contact Sources

Contact sources determine when contact objects need to be created, according to specific (stochastic) arrival processes. Each concrete arrival process must correspond to an algorithm for generating inter-arrival times. These times could depend on the entire state of the system in a complicated way, but they often depend only on the simulation time and previous inter-arrival times. Currently, only the Poisson process with piecewise-constant arrival rate and doubly stochastic variations of it described by Avramidis, Deslauriers, and L'Ecuyer (2004) are implemented, but many other processes could be added. For each process, the first arrival is scheduled when the arrival process is started, often at the beginning of the simulation.

The *factory* design pattern is used to allow the sources to construct contacts without knowing their types explicitly. The `ContactFactory` interface specifies a method called `newInstance` returning a newly-constructed and configured contact object. A contact source can create contacts from any class that implements this interface simply by invoking this `newInstance` method. Thus, changing the type of contact (and the name of its explicit constructor) requires no change to the implementation of the contact source.

When a new contact occurs, it is instantiated by the associated factory and broadcast to the registered *new-contact listeners*. Then the next arrival is scheduled. Each contact source is assigned a factory that typically constructs contacts of a single type. All contact sources can be initialized, started, and stopped.

A *predictive dialer* is normally used to generate outbound calls. The dialer's policy determines the number of calls to try on each occasion (as a function of the system's state), and supplies a list to extract them from. This list could be produced by a contact factory and is often assumed to be infinite for simplicity. Such lists could also be constructed from customer contacts who left a message, who were disconnected, etc.

For each call extracted from the dialer list, a success test is performed. This test succeeds with a probability being fixed or depending on the tested call, and the state of the system. Successful calls represent right party connects whereas failed calls represent wrong party connects and connection failures. The dialer generates a random delay representing the time between the beginning of dialing and the success or failure. This delay may depend on the success indicator, the call itself, the current time, etc. An event for broadcasting the call to registered listeners is then scheduled to occur at the time of success or failure.

The dialer defines separate lists of new-contact listeners for right party connects, and failed calls. Usually, only right party connects reach the router, but statistical collectors may need to listen to failed calls as well.

2.4 Waiting Queues

A `WaitingQueue` represents a data structure whose elements are waiting contacts. To support abandonment, rather than contact objects, the queue contains events being scheduled to happen at the time of automatic removal, e.g., abandonment, disconnection, etc. After a contact is added at the end of the queue, its dequeue event is constructed, and scheduled if a maximal queue time is available. Queued contacts can also be removed manually, e.g., by the router when the service can begin, or enumerated sequentially.

A registered *waiting-queue listener* can be notified about added and removed contacts. All the information being transmitted through the dequeue event, the listener interface remains unchanged even if new attributes need to be added to the event in the future. The reason of the removal is available for listeners through an integer called the *dequeue type*, encapsulated in the dequeue event. For example, this permits statistical collectors to distinguish abandonment from disconnection.

Two data structures are available for storing queued contacts, each implemented in concrete subclasses of `WaitingQueue`. The standard waiting queue uses a linked list for First In First Out (FIFO) and Last In First Out (LIFO) queues. When the number of priorities is finite and small, priority queues can be implemented efficiently by combining several standard waiting queues. For complex priority schemes, the library provides a priority queue using a red black tree with a user-defined comparator specifying how to order pairs of contacts. A red black tree (Cormen, Leiserson, Rivest, and Stein 2001) is a binary tree with automatic balancing for more stable search speed. All these waiting queues use the Java Collections Framework, and allow the internal data structure to be replaced if needed.

2.5 Agent Groups

An *agent group* i, represented by an instance of `AgentGroup`, contains $N_i(t) \in \mathbb{N}$ members at simulation time t. Among these agents, $N_{\mathrm{i},i}(t)$ are idle, and $N_{\mathrm{b},i}(t)$ are busy. Since agents terminate their service before they leave, we can have $N_i(t) < N_{\mathrm{b},i}(t)$, in which case $N_{\mathrm{g},i}(t) = N_{\mathrm{b},i}(t) - N_i(t)$

ghost agents need to disappear after they finish their work. As a result, the true number of agents in a group i at time t is given by $N_i(t)+N_{g,i}(t)$. New contacts are not accepted by the group when $N_i(t) \leq N_{b,i}(t)$. Since $N_{b,i}(t)$ includes the ghost agents, we have

$$N_i(t)+N_{g,i}(t) = N_{b,i}(t)+N_{i,i}(t). \tag{1}$$

Some idle agents may be unavailable to serve contacts at some times during their shift. They can be taking unplanned breaks, going to the bathroom, etc. These details can be modeled in the simulation if the appropriate information is available. But in practice, they are often approximated by various models such as an efficiency factor $\varepsilon_i \in [0,1]$, which corresponds to the fraction of agents being effectively busy or available to serve contacts. If $N_{b,i}(t) = 0$, the number of free agents $N_{f,i}(t)$ available to serve contacts is given by $N_{f,i}(t) = \text{round}(\varepsilon_i N_i(t))$ where round$(\cdot)$ rounds its argument to the nearest integer. If $N_{b,i}(t) > 0$, the number of busy members of the group, $N_{b,i}(t) - N_{g,i}(t)$, needs to be subtracted to get $N_{f,i}(t)$. This yields:

$$\text{round}(\varepsilon_i N_i(t))+N_{g,i}(t) = N_{b,i}(t)+N_{f,i}(t). \tag{2}$$

If $\varepsilon_i = 1$, $N_{f,i}(t) = N_{i,i}(t)$ and we are back to (1). This elementary efficiency model is provided because it can be used without simulating individual agents. When agents are differentiated, other more complex and more realistic models can easily be implemented by manipulating the state of agents during simulation.

The service of a contact is divided in two steps. After communicating with a customer (first step), an agent can perform after-contact work (second step), e.g., update an account, take some notes, etc. After the first step, the contact may exit the system (and release the allocated communication channel if necessary), or be transferred to another agent. However, the agent becomes free only after the second step (if any) is over. The end of these steps is scheduled using a simulation event that contains additional information about the service. As for the waiting queue, service can be terminated automatically through the event or manually through methods of `AgentGroup`. Special indicators tell us which type of termination has occured for each step. These facilities are useful to construct contact centers supporting preemptive service. For example, when the router receives a new phone call, it can interrupt the work of an agent answering an e-mail. This e-mail, along with information on the remaining service time, can be stored in a waiting queue for the service to be resumed later. The termination-type indicators permit the router to differentiate service terminations from service interruptions for statistical collecting purposes.

Registered *agent-group listeners* can be notified when $N_i(t)$ changes, when a service starts, and when it ends. As with the waiting queue, the simulation event is used rather than a temporary object to transmit information.

By default, for better efficiency, an agent group does not contain an object for each agent, preventing the simulator from differentiating them. Individual agents can of course be simulated by creating groups with a single member, but regrouping the agents can be useful for more efficient routing. The subclass `DetailedAgentGroup` offers an implementation where each individual agent is a separate object with its own characteristics. Each such agent can be added to or removed from a group at any time during a simulation.

2.6 Routers

A *router*, called an *automatic call distributor* (ACD) for call centers, can be any class listening to new contacts, and assigning them to agent groups or adding them to waiting queues. The router listens to service terminations to assign queued contacts to free agents and to waiting queue events for statistical collection and overflow support.

The library provides the `Router` class as a convenience tool, but since the elementary components do not have information on the structure of the router, the user can implement his own routing facilities if needed. Several routing systems could exist in parallel in a single model. For example, the current router requires Java code for its logic, but new systems could be defined for a XML-based logic with routing scripts constructed using a GUI. The former solution is faster, but the latter may be easier to use.

For statistical collection, it is generally not sufficient to listen to end of services directly, because a contact can be handled by several agents before leaving the system. For contacts to be counted correctly, an *exited-contact listener* can be registered with a router which knows exactly when they abandon, are blocked, and are served. The default router implementation provides facilities to register new and exited contact listeners, connect waiting queues and agent groups, and helper methods for implementing routing policies. The routing policy itself must be implemented in a subclass by defining fields for the data and implementing or overriding methods for the routing logic. The router needs schemes for agent and contact selections, and it can optionally clear waiting queues when the contact center does not have idle or busy agents capable of serving the waiting contacts. Algorithms to process dequeued and served contacts may also be needed in complex systems supporting overflow or service by multiple agents.

The library provides a few predefined policies inspired from Whitt and Wallace (2004) and Koole, Pot, and Talim (2003). These policies do not cover all possible scenarios, but flexibility is achieved by allowing subclasses of `Router` to be created.

A first class of policies uses ordered lists as follows. For each contact type k, the *type-to-group map* defines an ordered list $i_{k,1}, i_{k,2}, \ldots$ of agent groups. For each agent group i, the *group-to-type map* defines an ordered list $k_{i,1}, k_{i,2}, \ldots$ of contact types. These lists indicate which agent groups can serve a contact of type k and which contact types can be served by agents in group i, respectively. The order of the elements can be used to define priorities.

In a second type of policy, a *ranks matrix* assigns a rank or priority $r(i,k)$ to contacts of type k served by agents in group i. If the rank is ∞, contacts of type k cannot be served by agents in group i. Otherwise, the smaller is $r(i,k)$, the higher is the priority of contact type k for agents in group i. This structure allows equal priorities to exist, and avoids consistency problems, but routing policies are more complex. When ranks are equal, a secondary algorithm must be used for tie breaking, reducing the performance of the simulator.

3 EXAMPLE OF A SIMULATOR

We present a small example to illustrate some of the basic tools provided by the *ContactCenters* library. We consider a center with three contact types, two agent groups, and three two-hour periods. Contacts arrive according to a Poisson process with randomized piecewise-constant arrival rate $B\lambda_{k,p}$ for contact type k during period p, where the $\lambda_{k,p}$ are constant while B is a gamma random variable with mean 1 and variance $1/\alpha_0$, which represents the busyness of the day (Avramidis, Deslauriers, and L'Ecuyer 2004). If $B > 1$, the arrival rate of contacts is higher than usual. If $B < 1$, it is lower than usual.

When a contact arrives, an agent is selected from a group depending on its type. Contacts of type 0 can only be served by agents in group 0 while contacts of type 2 can only be served by agents in group 1. Contacts of type 1 are served by agents in group 0, or agents in group 1 if no agents are free in group 0. Service times are i.i.d. exponential variables with mean $1/\mu_p$ for contacts arriving during period p.

Agents in each group i are not differentiated, and $N_i(t)$ changes between periods while being constant within each period. If $N_{\mathrm{b},i}(t) \geq N_i(t)$ at some time t, ongoing services are finished, but new contacts are not accepted until $N_{\mathrm{b},i}(t) < N_i(t)$.

A contact that cannot be served immediately is added to a waiting queue corresponding to its type. Abandonment is supported, with patience times that are i.i.d. exponentials with mean $1/\nu_p$ for contacts arriving during period p.

Suppose we are interested in the long-term overall service level and the occupancy ratio of the first agent group. These quantities are defined as follows. For a given constant $s > 0$ which can be interpreted as the maximum acceptable waiting time in the queue, let X be the total number of served contacts, $X_{\mathrm{g}}(s)$ the number of served contacts having waited less than s, Y the total number of contacts having abandoned, and $Y_{\mathrm{b}}(s)$ the number of contacts having abandoned after waiting at least s. The *service level* is defined as

$$g(s) = \frac{E[X_{\mathrm{g}}(s)]}{E[X + Y_{\mathrm{b}}(s)]}.$$

The *occupancy ratio* of an agent group i is defined as

$$o_i = \frac{E\left[\int_0^T N_{\mathrm{b},i}(t)\ dt\right]}{E\left[\int_0^T (N_i(t) + N_{\mathrm{g},i}(t))\ dt\right]},$$

where $T = t_{P+1}$ is the time at which all contacts are served after the end of the day. We also estimate $E[X_{\mathrm{g},k,p}(s)]$, the expected number of served contacts meeting the service level requirement, for contacts of type k arrived during period p, for each k and p.

Figure 1 presents the code implementing this small model. Its first part declares constants and variables, and creates objects to set up the program. The `main` method, located at the end of the second part, constructs a simulator using `new SimpleMSK()`, triggers the simulation by calling `simulate`, and displays statistical results by using `printStatistics`. The `simulate` method calls `simulateOneDay` n times while `simulateOneDay` initializes the system for a new replication, starts the simulation, and collects some observations.

For simplicity, parameters are encoded into constants, although real-life simulators should read them from files. The simulator declares the components of the contact center such as the arrival processes, agent groups, waiting queues, and router, which do not compute any statistic. Counters and statistical collectors are declared separately to estimate the performance measures of interest only. For X, $X_{\mathrm{g}}(s)$, Y, and $Y_{\mathrm{b}}(s)$, simple integers are sufficient for this example, but a matrix is needed for $X_{\mathrm{g},k,p}(s)$. The *counters* are used to compute per-replication values whereas the *statistical probes* collect these values to get averages, variances, and confidence intervals across replications.

The constructor `SimpleMSK()`, at the bottom of the first page of the program, creates the components declared in fields, and links them together. The period-change event is constructed with $P+2 = 5$ periods, i.e., one preliminary period, three main periods, and one wrap-up period. Since $t_0 = 0$, the preliminary period has a duration of 0.

For each contact type, a factory and an arrival process are constructed. The arrival process automatically registers as a period-change listener to be notified when a new period starts. This will allow the arrival rate to be automatically changed from period to period. The busyness generator `bgen` is then constructed for generating gamma variates using inversion. Constructing the agent groups requires the

```
// Import declarations

public class SimpleMSK {
   // All times are in minutes
   static final int K                   = 3;          // Number of contact types
   static final int I                   = 2;          // Number of agent groups
   static final int P                   = 3;          // Number of periods
   static final double PERIODDURATION   = 120.0;      // Two hours
   // LAMBDA[k][p] gives the arrival rate for type k in period p
   static final double[][] LAMBDA       =
      { { 0, 4.2, 5.3, 3.2, 0 }, { 0, 5.1, 4.3, 4.8, 0 }, { 0, 6.3, 5.2, 4.8, 0 } };
   static final double ALPHA0           = 28.7;           // Gamma param. for busyness
   static final double[] MU = { 0.5, 0.5, 0.6, 0.4, 0.4 }; // Service rate for each period
   static final double[] NU = { 0.3, 0.3, 0.4, 0.2, 0.2 }; // Abandonment rate for each period
   static final double AWT              = 20/60.0;        // Acceptable waiting time (20 sec.)
   // NUMAGENTS[i][p] gives the number of agents for group i in period p
   static final int[][] NUMAGENTS       = { { 0, 12, 18, 9, 9 }, { 0, 15, 20, 11, 11 } };
   // Routing table, TYPETOGROUPMAP[k] and GROUPTOTYPEMAP[i] contain ordered lists
   static final int[][] TYPETOGROUPMAP  = { { 0 }, { 0, 1}, { 1 } };
   static final int[][] GROUPTOTYPEMAP  = { { 1, 0 }, { 2, 1 } };
   static final double LEVEL            = 0.95;       // Level for confidence intervals
   static final int NUMDAYS             = 10000;      // Number of replications

   PeriodChangeEvent pce;                // Event marking the beginning of each period
   PiecewiseConstantPoissonArrivalProcess[] arrivProc
      = new PiecewiseConstantPoissonArrivalProcess[K];
   AgentGroup[] groups = new AgentGroup[I];
   WaitingQueue[] queues = new WaitingQueue[K];
   Router router;
   RandomVariateGen sgen;                             // Service times generator
   RandomVariateGen pgen;                             // Patience times generator
   RandomVariateGen bgen;                             // Busyness  generator

   // Counters
   int numGoodSL, numServed, numAbandoned, numAbandonedAfterAWT;
   double[][] numGoodSLKP = new double[K][P];
   GroupVolumeStat vstat;                      // Integral of the occupancy ratio

   // statistical collectors
   Tally served = new Tally ("Number of served contacts");
   Tally abandoned = new Tally ("Number of contacts having abandoned");
   MatrixOfTallies goodSLKP = new MatrixOfTallies
      ("Number of contacts meeting target service level",
       new String[] { "Type 0", "Type 1", "Type 2" },
       new String[] { "Period 0", "Period 1", "Period 2" });
   RatioTally serviceLevel = new RatioTally ("Service level");
   RatioTally occupancy = new RatioTally ("Occupancy ratio");

   SimpleMSK() {
      // One dummy preliminary period, P main periods, and one wrap-up period,
      // main periods start at time 0.
      pce = new PeriodChangeEvent (PERIODDURATION, P + 2, 0);
      for (int k = 0; k < K; k++)                     // For each contact type
         arrivProc[k] = new PiecewiseConstantPoissonArrivalProcess
            (pce, new MyContactFactory (k), LAMBDA[k], new MRG32k3a());
      bgen = new GammaGen (new MRG32k3a(), new GammaDist (ALPHA0, ALPHA0));
      for (int i = 0; i < I; i++) groups[i] = new AgentGroup (pce, NUMAGENTS[i]);
      for (int q = 0; q < K; q++) queues[q] = new StandardWaitingQueue();
      sgen = MultiPeriodGen.createExponential (pce, new MRG32k3a(), MU);
      pgen = MultiPeriodGen.createExponential (pce, new MRG32k3a(), NU);
      router = new SingleFIFOQueueRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
      for (int k = 0; k < K; k++) arrivProc[k].addNewContactListener (router);
      for (int i = 0; i < I; i++) router.setAgentGroup (i, groups[i]);
      for (int q = 0; q < K; q++) router.setWaitingQueue (q, queues[q]);
      router.addExitedContactListener (new MyContactMeasures());
      vstat = new GroupVolumeStat (groups[0]);
   }
```

Figure 1: Example of A Contact Center Simulator

```
   // Creates the new contacts
   class MyContactFactory implements ContactFactory {
      int type;
      MyContactFactory (int type) { this.type = type; }
      public Contact newInstance() {
         Contact contact = new Contact (type);
         contact.setDefaultServiceTime (sgen.nextDouble());
         contact.setDefaultPatienceTime (pgen.nextDouble());
         return contact;
      }
   }

   // Updates counters when a contact exits
   class MyContactMeasures implements ExitedContactListener {
      public void blocked (Router router, Contact contact, int bType) {}
      public void dequeued (Router router, WaitingQueue.DequeueEvent ev) {
         ++numAbandoned;
         if (ev.getContact().getTotalQueueTime() >= AWT) ++numAbandonedAfterAWT;
      }
      public void served (Router router, AgentGroup.EndServiceEvent ev) {
         ++numServed;
         Contact contact = ev.getContact();
         if (contact.getTotalQueueTime() < AWT) {
            ++numGoodSL;
            int period = pce.getPeriod (contact.getArrivalTime()) - 1;
            if (period >= 0 || period < P) ++numGoodSLKP[contact.getTypeId()][period];
         }
      }
   }

   void simulateOneDay() {
      Sim.init();    pce.init();
      double b = bgen.nextDouble();   // Busyness factor for today
      for (int k = 0; k < K; k++) arrivProc[k].init (b);
      for (int i = 0; i < I; i++) groups[i].init();
      for (int q = 0; q < K; q++) queues[q].init();
      numGoodSL = numServed = numAbandoned = numAbandonedAfterAWT = 0;    vstat.init();
      for (int k = 0; k < K; k++) for (int p = 0; p < P; p++) numGoodSLKP[k][p] = 0;
      for (int k = 0; k < K; k++) arrivProc[k].start();
      pce.start();    Sim.start();    // Simulation runs here
      pce.stop();
      served.add (numServed);  abandoned.add (numAbandoned);  goodSLKP.add (numGoodSLKP);
      serviceLevel.add (numGoodSL, numServed + numAbandonedAfterAWT);
      double Nb = vstat.getStatNumBusyAgents().sum();   // Integral of N_b0(t)
      double N = vstat.getStatNumAgents().sum();        // Integral of N_0(t)
      double Ng = vstat.getStatNumGhostAgents().sum();  // Integral of N_g0(t)
      occupancy.add (Nb, N + Ng);
   }

   // Simulate n independent days
   void simulate (int n) {
      served.init();  abandoned.init();  goodSLKP.init();
      serviceLevel.init();    occupancy.init();
      for (int r = 0; r < n; r++) simulateOneDay();
   }

   public void printStatistics() {
      System.out.println (served.reportAndCIStudent (LEVEL, 3));
      System.out.println (abandoned.reportAndCIStudent (LEVEL, 3));
      System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
      System.out.println (occupancy.reportAndCIDelta (LEVEL, 3));
      for (int k = 0; k < K; k++)
         System.out.println (goodSLKP.rowReportAndCIStudent (k, LEVEL, 3));
   }

   public static void main (String[] args) {
      SimpleMSK s = new SimpleMSK();  s.simulate (NUMDAYS);  s.printStatistics();
   }
}
```

Figure 1: Example of A Contact Center Simulator (continued)

period-change event, and an array containing the number of agents for each period. Each agent group also registers as a period-change listener for $N_i(t)$ to be automatically updated during the simulation. During the preliminary period, $N_i(t) = 0$ for all i, while during the wrap-up period, $N_i(t)$ corresponds to the number of agents in the last main period. A second constructor is available to create an agent group not using a period-change event, for which $N_i(t)$ must be changed manually.

Service and patience times are generated using `sgen` and `pgen` which are random variate generators for multiple periods. Such generators use a period-change event to determine the current period and selects a period-specific generator to get random values. The generic way for constructing them is to create a random variate generator for each period and give the array of generators, with a period-change event, to the constructor of `MultiPeriodGen`. For some distributions such as exponential, helper methods such as `createExponential` are available to construct the generators more conveniently; this method is used in the constructor to initialize `sgen` and `pgen`.

For the router to be constructed, a type-to-group map and a group-to-type map are needed. The selected `SingleFIFOQueueRouter` class affects how these structures are used. Note how the arrival processes, the waiting queues, and the agent groups are linked to the router. An exited-contact listener is also connected for statistical collection.

The `vstat` object is used for computing the integrals needed for the occupancy ratio in the first agent group. It internally registers as an agent-group listener to observe and integrate $N_i(t)$, $N_{\mathrm{g},i}(t)$, $N_{\mathrm{f},i}(t)$, $N_{\mathrm{i},i}(t)$, and $N_{\mathrm{b},i}(t)$. Although this is not used in this example, it can also compute $N_{\mathrm{b},i,k}(t)$, the number of busy agents in group i serving contacts of type k, if K is given to the constructor. The program could also compute the occupancy ratio in the second agent group as well as the overall occupancy ratio.

The heart of the program is the `simulateOneDay` method located in the middle of the second page. It first initializes the simulation clock and the period-change event. All contact center elements are then initialized to eliminate any side effect from previous replications. The arrival processes are initialized with a busyness factor to randomize the arrival rates. All the statistical counters are reset to 0, and the volume calculator is reset, which initializes the internal "accumulators" that compute the integrals. Starting the arrival processes using `start` schedules the first arrivals. The period-change event is started, scheduling an event at time 0 for the first main period, and the simulator is started using `Sim.start`, which starts executing events.

When an arrival process triggers an arrival, the `newInstance` method implemented in `MyContactFactory`, shown at the top of the second page of the program, is called on the corresponding contact factory. One factory object has been constructed for each arrival process, the only difference between them being the value of the `type` field. The factory constructs a contact of the appropriate type and generates a service time and a patience time. Each random value is associated with the returned contact object. The arrival process broadcasts the contact to the router, generates a new arrival time and schedules the next arrival.

When a contact of type 0 arrives, the router takes the element 0 of the type-to-group map, which corresponds to an ordered list containing the agent group 0 only. If $N_{\mathrm{f},0}(t) > 0$, the contact is served immediately. Otherwise, it is added to waiting queue 0. Contacts of type 2 are treated similarly. For contacts of type 1, the router obtains an ordered list containing 0 and 1. If $N_{\mathrm{f},0}(t) > 0$, the contact is served immediately. Otherwise, it overflows to the next agent group in the list. If $N_{\mathrm{f},1}(t) > 0$, the contact is served. Otherwise, it is added at the end of queue 1.

When an agent within group 0 becomes free or is added, the router uses the group-to-type map to obtain its ordered list, $\{1,0\}$. The chosen router selects the queued contact with the longest waiting time rather than using the order induced by the list. The longest waiting time is used because of the selected routing policy; by using a different policy, i.e., a different subclass of `Router`, another selection rule could be enforced. If the waiting queues accessible for agents in group 0 contain no contact, the agent remains free until new arrivals occur. Agents in group 1 have similar contact selection rules.

Each contact exiting the system is notified to the registered exited-contact listener. The `blocked` method does nothing because the capacity of the contact center is infinite by default. When a contact leaves the queue without service, a new abandonment is counted. If its waiting time is greater than or equal to s, an abandonment after the acceptable waiting time is also counted. When a contact is served, a new service is counted. If its waiting time is small enough, it is also counted as a good contact, i.e., a contact meeting service level requirement.

For a good contact to be counted in `numGoodSLKP`, the main period of its arrival must be determined. The `getPeriod` method returns a value in the range $1,\ldots,P$ which is converted to a main period index by subtracting 1. If the main period index is negative or greater than or equal to P, the arrival occurred during the preliminary or wrap-up periods, and the event is ignored. Otherwise, the appropriate element of the matrix is incremented.

The contact center closes at time $t_P = 120P = 360$ (after 6 hours of operation). Since the arrival rates $\lambda_{k,P+1}$ are 0 for all k, the arrival processes stop automatically at the beginning of the wrap-up period. All queued contacts are then served before the simulation stops.

Since the end of the wrap-up period is not scheduled as an event, the `stop` method is used to notify registered period-change listeners after `Sim.start()` returns. Com-

puted observations are added to collectors and the service level and occupancy ratio are computed for the replication.

If occupancy ratio was estimated for opening hours only, the integrals would have to be obtained at time t_P rather than time T; this requires a custom period-change listener, which we avoided to keep the program as simple as possible.

4 SPEED COMPARISON WITH A COMMERCIAL PRODUCT

To evaluate the performance of the *ContactCenters* library, we compare it with Rockwell's *Arena Contact Center Edition 8.0* (Rockwell Automation, Inc. 2005), using four models provided as examples with the latter commercial product. We provide a brief summary of these four models. More details can be found in the Arena Contact Center Edition User's Guide. In all examples, arrivals follow a Poisson process with a constant arrival rate through all the simulation, except for the first (main) period.

Telethon deals with the organization of a pledge drive local public radio station. From 6AM to 10AM, volunteers process contacts to manage donations. Donors have the possibility to abandon or being disconnected and asked to leave a message.

Bilingual represents a contact center serving an English and a Spanish populations. English-speaking, Spanish-speaking, and bilingual agents are available to serve the contacts. The system is slightly more complex than Telethon, because customers have the option to contact back and are only routed to agents capable of serving them. However, specialists do not have priority over bilingual agents.

Bank represents a bank model where each agent can process all contact types but handles its specialty more efficiently. This multi-skill contact center models agents' preferences and has approximately the same complexity level as Bilingual.

Teamwork models a contact center with complex routing logic in which a contact is processed by several agents. Many customers abandon after waiting for a receptionist while many others are disconnected when trying to reach technical support. Some agents are required to perform after-contact work after the served contact is transferred. Although this model supports a single contact type, it is more complex than the three other ones, since contacts are served by multiple agents.

Each of these four models has been implemented with *ContactCenters* and simulated for $n = 1000$ independent replications. CPU times have been obtained using facilities from SSJ. The four examples were also executed in Arena, in *batch mode*, to get the fastest possible execution times. Since Arena does not compute the execution time of a model directly, an external program executing the models through Component Object Model (COM) was used to get the system time which approximates the CPU time. For maximal accuracy of the system times, no other user-level tasks than the Arena simulation were performed on the machine during the tests.

Table 1 compares the performance of the *ContactCenters* samples with that of Arena. For each table entry, we find the required CPU time on the left and the number of contacts processed per second on the right. The reported times are computed on an AMD Athlon Thunderbird 1000MHz. Java times are computed under Linux, using Sun Java Runtime Environment (JRE) 1.4.2 and 1.5.0 while Arena times are computed under Microsoft Windows XP. To approximate the number of contacts per second, the model-dependent expected number of arrivals over all replications, $nE[A]$, is divided by the estimated CPU time.

ContactCenters runs approximately 25 times faster than Arena on these examples. The execution times generally increase with the complexity of the model. The Teamwork model is more complex than the other examples, but it runs faster than Telethon under Arena and faster than Bank under *ContactCenters*. The explanation is that in Teamwork, the abandonment rate is very high, because contacts are filtered by the two-servers queue modeling the receptionists, and contacts directed to technical support are disconnected if no agent is available. The processing time of an abandoned contact is smaller, because the service requires scheduling an extra event or allocating and releasing a resource.

For both systems, performance depends on the number of contacts to be processed as well as their path into the system. It also depends on the routing policy being used, whose performance depends on the size and complexity of the contact center.

5 CONCLUSION

The library ContactCenters is flexible enough to simulate practically any model of a contact center using Java and SSJ. Some examples from the Arena User's Guide have been easy to implement and they execute faster than with the commercial tool.

In the future, we plan to experiment with variance reduction techniques that could improve simulation efficiency. We also plan to test various subgradients computation methods for optimization. The current generic simulator using XML for parameter files will be maintained, and new ones may be constructed for other contact center designs.

ACKNOWLEDGMENTS

This research was supported by grants number OGP-0110050 and CRDPJ-251320 from NSERC-Canada, a grant from Bell Canada via the Bell University Laboratories, and grant number 00ER3218 from NATEQ-Québec to the second author. We thank Athanassios Avramidis, Mehmet

Table 1: Performance of the ContactCenters Library Compared with Arena

Example	$E[A]$	Arena		JRE1.4		JRE1.5	
Telethon	1000	4m23s	3802/s	10s	103950/s	10s	102040/s
Bilingual	5000	23m39s	3523/s	48s	104866/s	44s	112969/s
Bank	3600	23m57s	2505/s	48s	75774/s	46s	78947/s
Teamwork	7000	22m56s	5087/s	1m23s	83923/s	1m23s	84592/s

Tolga Ceẓik and Wyean Chan for their helpful comments on the design of ContactCenters classes, and for testing the constructed simulators during software development.

REFERENCES

Atlason, J., M. A. Epelman, and S. G. Henderson. 2003. Using simulation to approximate subgradients of convex performance measures in service systems. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 1824–1832: IEEE Press.

Atlason, J., M. A. Epelman, and S. G. Henderson. 2004. Call center staffing with simulation and cutting plane methods. *Annals of Operations Research* 127:333–358.

Avramidis, A. N., A. Deslauriers, and P. L'Ecuyer. 2004. Modeling daily arrivals to a telephone call center. *Management Science* 50 (7): 896–908.

Bapat, V. 2003. The arena product family: Enterprise modeling solutions. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 210–217: IEEE Press.

Buist, E. 2005. Conception et implantation d'une librairie pour la simulation de centres de contacts. Master's thesis, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal. Forthcoming.

Buist, E., and P. L'Ecuyer. 2005. *ContactCenters: A Java library for simulating contact centers*. Software user's guide, forthcoming.

Ceẓik, M. T., and P. L'Ecuyer. 2004. Staffing multiskill call centers via linear programming and simulation. submitted.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. 2001, September. *Introduction to algorithms*. second ed. MIT Press.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1998. *Design patterns: Elements of reusable object-oriented software*. second ed. Reading, Mass.: Addison-Wesley.

Gans, N., G. Koole, and A. Mandelbaum. 2003. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing and Service Operations Management* 5:79–141.

Koole, G., A. Pot, and J. Talim. 2003. Routing heuristics for multi-skill call centers. In *Proceedings of the 2003 Winter Simulation Conference*, 1813–1816: IEEE Press.

L'Ecuyer, P. 2004. *SSJ: A Java library for stochastic simulation*. Software user's guide, Available at ⟨http://www.iro.umontreal.ca/~lecuyer⟩.

L'Ecuyer, P., and E. Buist. 2005. Simulation in Java with SSJ. In *Proceedings of the 2005 Winter Simulation Conference*. submitted.

L'Ecuyer, P., L. Meliani, and J. Vaucher. 2002. SSJ: A framework for stochastic simulation in Java. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, 234–242: IEEE Press.

Mehrotra, V., and J. Fama. 2003. Call center simulation modeling: Methods, challenges, and opportunities. In *Proceedings of the 2003 Winter Simulation Conference*, 135–143: IEEE Press.

NovaSim 2003. ccProphet — simulate your call center's performance. See ⟨http://www.novasim.com/CCProphet/⟩.

Rockwell Automation, Inc. 2005. Arena simulation. See ⟨http://www.arenasimulation.com⟩.

Whitt, W., and R. B. Wallace. 2004. A staffing algorithm for call centers with skill-based routing. working paper, available at ⟨http://www.columbia.edu/~ww2040/poolingMSOMrevR.pdf⟩.

AUTHOR BIOGRAPHIES

ERIC BUIST is a M.Sc. Student at the Université de Montréal. His main interests are software engineering, object-oriented programming, and simulation. His e-mail address is <buisteri@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. 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 obtained the prestigious *E. W. R. Steacie* fellowship in 1995-97 and a *Killam* fellowship in 2001-03. His recent research articles are available on-line from his web page: ⟨http://www.iro.umontreal.ca/~lecuyer⟩.