

# jMarkov User's Guide

Germán Riaño, Julio Góez, and Juan F. Pérez

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Building Large - Scale Markov Chains</b>	<b>2</b>
2.1	Space state building algorithm . . . . .	2
2.2	Measures of Performance . . . . .	3
<b>3</b>	<b>Framework Design</b>	<b>4</b>
3.1	Java and Object Oriented Programming . . . . .	4
3.2	Build Package . . . . .	4
3.3	Basic Package . . . . .	4
3.4	The Solvers Package . . . . .	6
<b>4</b>	<b>Examples</b>	<b>6</b>
4.1	Example: An M/M/2/N with different servers . . . . .	6
4.1.1	The model . . . . .	6
4.1.2	Class QueueMM2dNState . . . . .	7
4.1.3	Class QueueMM2dN . . . . .	7
4.1.4	Code . . . . .	7
4.2	Multiple Server Queue . . . . .	11
4.2.1	The model . . . . .	11
4.2.2	Code . . . . .	12
4.3	Drive Thru . . . . .	17
4.3.1	Code . . . . .	17
4.3.2	Results . . . . .	24
<b>5</b>	<b>Modeling Quasi-Birth and Death Processes</b>	<b>25</b>
5.1	Quasi-Birth and Death Processes . . . . .	25
5.2	Measures of performance for QBDs . . . . .	26
5.3	Modeling QBD with jQBD . . . . .	26
5.4	An Example . . . . .	27
<b>6</b>	<b>Modeling Priority Queues: incorporating phase-type distributions with jPhase</b>	<b>28</b>
<b>7</b>	<b>Further Development</b>	<b>32</b>
	<b>References</b>	<b>32</b>

# 1 Introduction

The main purpose of jMarkov is facilitating the development and application of large scale Markovian models, so that they can be used by engineers with basic programming and stochastic modeling skills.

The project is composed of four modules: jMarkov, jQBD, jPhase, and jMDP. This focuses on jMarkov and jQBD, which are used to build Markov Chains and Quasi-Birth and death processes (QBD). The other two modules have their own manuals.

With jPhase a user can easily manipulate Phase-Type distributions (PH). These distributions are quite flexible and powerful, and a model that is limited to PH in practical terms can model many situations. For details see [9] and [8].

jMDP is used to build and solve Markov Decision Process (MDP). MDP, or, as is often called, Probabilistic Dynamic Programming allows the analyst to design optimal control rules for a Markov Chain. jMDP works for discrete and continuous time MDPs. For details see [12] and [11]

For up-to date information, downloads and examples check jMarkov's website at <https://projects.coin-or.org/jMarkov/>.

## 2 Building Large - Scale Markov Chains

In this section, we will describe the basic algorithms used by jMarkov to build Markov Chains. Although we limit our description to Continuous Time Markov Chain (CTMC), jMarkov can handle also Discrete Time Markov Chains (DTMC).

Let  $\{X(t), t \geq 0\}$  be a CTMC, with finite space state  $\mathcal{S}$  and generator matrix  $\mathbf{Q}$ , with components

$$q_{ij} = \lim_{t \downarrow 0} P \{X(t) = j | X(0) = i\} \quad i, j \in \mathcal{S}.$$

It is well known that this generator matrix, along with the initial conditions, completely determines the transient and stationary behavior of the Markov Chain (see, e.g, [5]). The diagonal components  $q_{ii}$  are non-positive and represent the exponential holding rate for state  $i$ , whereas the off diagonal elements  $q_{ij}$  represent the transition rate from state  $i$  to state  $j$ .

The transient behavior of the system is described by the matrix  $\mathbf{P}(t)$  with components

$$p_{ij}(t) = P \{X(t+s) = j | X(s) = i\} \quad i, j \in \mathcal{S}.$$

This matrix can be computed as

$$\mathbf{P}(t) = e^{\mathbf{Q}t} \quad t > 0.$$

For an irreducible chain, the stationary distribution  $\boldsymbol{\pi} = [\pi_1, \pi_2, \dots]$  is determined as the solution to the following system of equations

$$\begin{aligned} \boldsymbol{\pi} \mathbf{Q} &= \mathbf{0} \\ \boldsymbol{\pi} \mathbf{1} &= 1, \end{aligned}$$

where  $\mathbf{1}$  is a column vector of ones.

### 2.1 Space state building algorithm

Transitions in a CTMC are triggered by the occurrence of events such as arrivals and departures. The matrix  $\mathbf{Q}$  can be decomposed as  $\mathbf{Q} = \sum_{e \in \mathcal{E}} \mathbf{Q}^{(e)}$ , where  $\mathbf{Q}^{(e)}$  contains the transition rates associated with event  $e$ , and  $\mathcal{E}$  is the set of all possible events that may occur. In large systems, it is not easy to know in advance how many states there are in the model. However, it is possible to determine what events occur in every state, and the destination states produced by each transition

when it occurs. jMarkov works based on this observation, using an algorithm similar to the algorithm buildRS presented by Ciardo [1]; see Figure 1. The algorithm builds the space state and the transition rate by a deep exploration of the graph. It starts with an initial state  $i_0$  and searches for all other states. At every instant, it keeps a set of “unchecked” states  $\mathcal{U}$  and the set of states  $\mathcal{S}$  that have been already checked. For every unchecked state the algorithm finds the possible destinations and, if they had not been previously found, they are added to the  $\mathcal{U}$  set. To do this, it first calls the function **active** that determines if an event can occur. If it does, then the possible destination states are found by calling the function **dests**. The transition rate is determined by calling the function **rate**. From this algorithm, we can see that a system is fully described once the states and events are defined and the functions **active**, **dests**, and **rate** have been specified. As we will see, modeling a problem with jMarkov entails coding these three functions.

```

 $\mathcal{S} = \emptyset, \mathcal{U} = \{i_0\}, \mathcal{E}$  given.
while  $\mathcal{U} \neq \phi$  do
  for all  $e \in \mathcal{E}$  do
    if active( $i, e$ ) then
       $\mathcal{D} := \text{dests}(i, e)$ 
      for all  $j \in \mathcal{D}$  do
        if  $j \notin \mathcal{S} \cup \mathcal{U}$  then
           $\mathcal{U} := \mathcal{U} \cup \{j\}$ 
        end if
       $R_{ij} := R_{ij} + \text{rate}(i, j, e)$ 
    end for
  end if
end for
end while

```

Figure 1: BuildRS algorithm

## 2.2 Measures of Performance

When studying Markovian systems, the analyst is usually interested in the transient and steady state behavior of measures of performance (MOPs). This is accomplished by attaching rewards to the model. Let  $\mathbf{r}$  be a column vector such that  $r(i)$  represents the expected rate at which the system receives rewards whenever it is in state  $i \in \mathcal{S}$ . Here the term *reward* is used for any measure of performance that might be of interest, not necessarily monetary. For example, in queueing systems  $r(i)$  might represent the number of entities in the system, or the number of busy servers, when the state is  $i$ . The expected reward rate at time  $t$  is computed according to

$$E(r(X(t))) = \mathbf{aP}(t)\mathbf{r},$$

where the row vector  $\mathbf{a}$  has the initial conditions of the process (i.e.,  $a_i = P\{X(0) = i\}, i \in \mathcal{S}$ ). Similarly, for an irreducible CTMC, the long run rate at which the system receives rewards is calculated as

$$\lim_{t \rightarrow \infty} \frac{1}{t} \int_0^t E(r(X(s))) ds = \boldsymbol{\pi} \mathbf{r}.$$

As we will see, jMarkov provides mechanisms to define this type of rewards and can compute both, transient and steady state MOPs. There are other type of rewards, like expected time in the system, which can be easily computed using Little law.

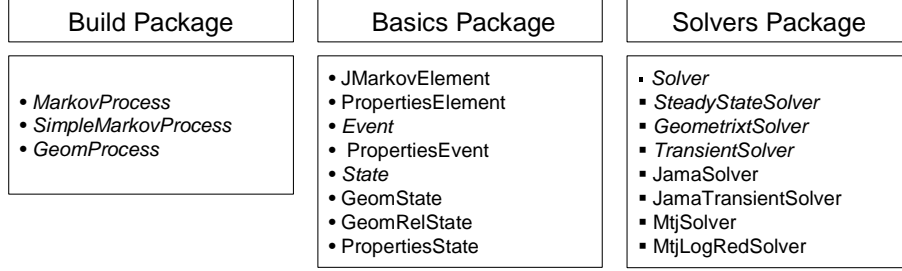


Figure 2: Class classification

### 3 Framework Design

In this section, we give a brief description of jMarkov’s framework architecture. We start by describing object-oriented programming and then describe the three packages that compose jMarkov.

#### 3.1 Java and Object Oriented Programming

Java is a programming language created by Sun Microsystems [13]. The main characteristics that Sun intended to have in Java are: Object-Oriented, robust, secure, architecture neutral, portable, high performance, interpreted, threaded and dynamic.

Object-Oriented Programming (OOP) is not a new idea. However, it did not have an increased development until recently. OOP is based on four key principles: abstraction, encapsulation, inheritance and polymorphism. An excellent explanation of OOP and the Java programming language can be found in [14].

The abstraction capability is the one that interests us most. Java allows us to define abstract types like `MarkovProcess`, `State`, etc. We can also define *abstract* functions like `active`, and `deaths`. We can program the algorithm in terms of these abstract objects and functions and the program works independently of the particular implementation of the aforementioned elements. All the user has to do is to *implement* the abstract functions. What is particularly nice is that if a function is declared as abstract, then the compiler itself will force the user to implement it before she attempts to run the model.

#### 3.2 Build Package

The build package is the main one in jMarkov since it contains the classes that take care of building the state space and transition matrices. The main classes are `MarkovProcess`, `SimpleMarkovProcess`, and `GeomProcess` (see Figure 3). Whereas the first two allow to model general Markov processes, `GeomProcess` is used for Quasi-Birth and Death Processes (QBD) and its description is given in Section 5.3 below.

The class `SimpleMarkovProcess` represents a Markov chain process, and contains three abstract methods that implement the three aforementioned functions in the algorithm BuildRS: `active`, `deaths`, and `rate`. In order to model a problem the user has to extend this class and implement the three functions. An example is given in Section 5.4. The class `MarkovProcess` is the main class in the module, and provides a more general mechanism to describe the dynamics of the system. It also contains tools to communicate with the solvers to compute steady state and transient solutions, and print them in a diverse array of ways. For details, see [10].

#### 3.3 Basic Package

This package contains the building blocks needed to describe a Markov Chain. It contains classes such as `State`, and `Event`, which allow the user to code a description of the states and events,

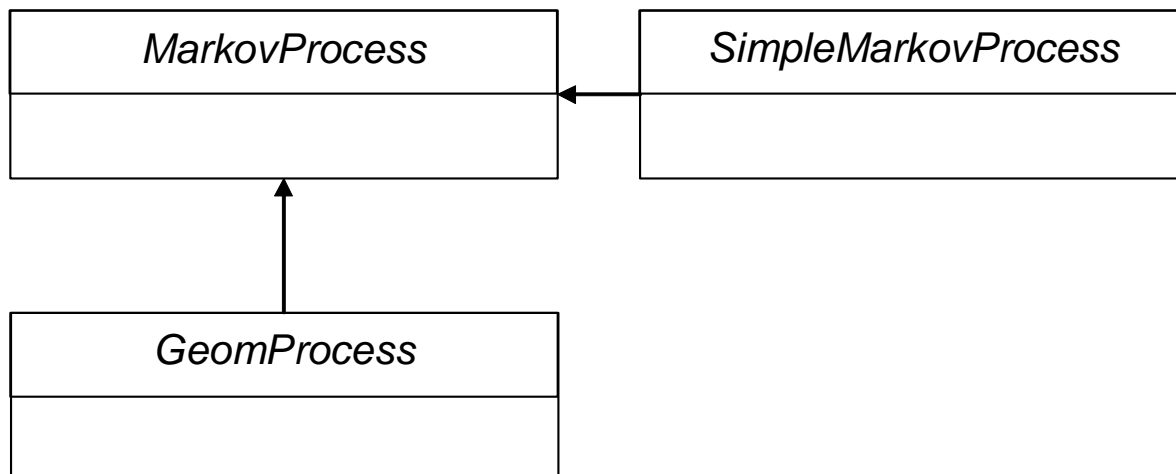


Figure 3: Class diagram build module

respectively (see Figure 4). The user has freedom to choose any particular coding that best describes the states in her model, like any combination of integers, strings, etc. However, she must establish a complete ordering among the elements since, for efficiency, jMarkov works with ordered sets. For simplicity, however, a built-in class is provided, called **PropertiesState**, that describes the state with an array of integers, something which is quite appropriate for many applications. Similarly, there is an analogous class called **PropertiesEvent**. The package also contains the classes **States** and **Events** that are used to describe collections of states and events. These are fairly general classes, since all that is required from the user is to provide a mechanism to “walk through” the elements of the set, taking advantage of Java iterator mechanism. This implies that, for large sets, there is no need to generate (and store) all the elements in the set. For convenience, the package provides implementations of these set classes based on sorted sets classes available in Java.

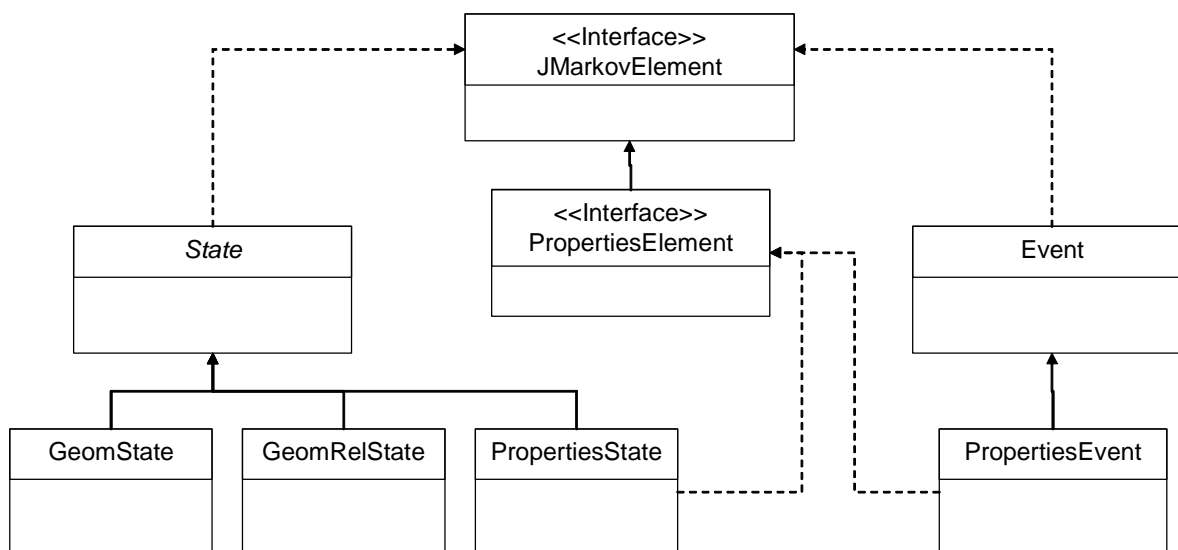


Figure 4: Class diagram for the basic package

### 3.4 The Solvers Package

As stated above, jMarkov separates modeling from solving. Various solvers are provided to find steady-state and transient probabilities (see Figure 5). If the user does not specify the solver to use, one is provided by default. However, the architecture is flexible enough to allow an interested user to choose a different solver, or, if she desires, to implement her own. The basic class is called **Solver**, that has two sub-classes called **SteadyStateSolver**, **TransientSolver**, and **GeomSolver** (see Figure 5). As the names indicate, the first two provide solvers for steady state and transient probabilities, whereas the latter is used for QBDs, as explained in section 5. The implementations provided rely on two popular Java packages to handle matrix operations JAMA [3] and MTJ [2], for dense and sparse matrices, respectively.

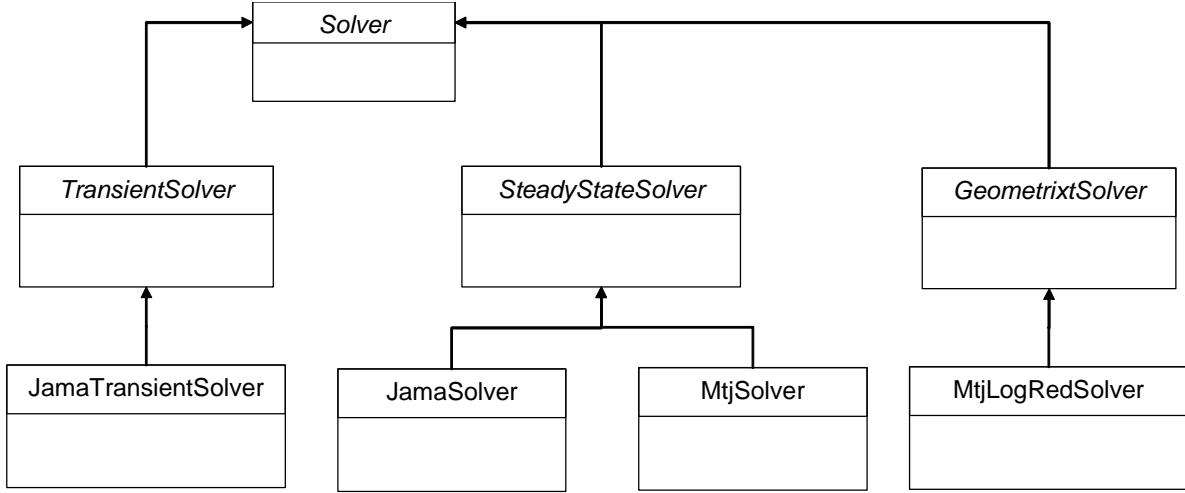


Figure 5: Class diagram of the solvers package

## 4 Examples

### 4.1 Example: An M/M/2/N with different servers

Assume that a system has Poisson arrivals with rate  $\lambda$ . There are two exponential servers with rates  $\mu_1$  and  $\mu_2$  respectively. There is a maximum of  $N$  customers in the system. An arriving customer that finds the system empty will go to server 1 with probability  $\alpha$ . Otherwise he will pick the first available server, or join a single FCFS queue. If there are  $N$  in the system the customer goes away.

#### 4.1.1 The model

We model this system with the triple  $\mathbf{X}(t) = (X(t), Y(t), Z(t))$ , where  $X(t)$  and  $Y(t)$  represents the status of the server (1 if busy 0 otherwise) and  $Z(t)$  represents the number in queue, which is a number from 0 to  $N - 2$ . There are  $2 \times 2 \times N - 2$  potential states, however not all combinations of  $X, Y$  and  $Z$  are possible. For example the state  $(0, 1, 2)$  is not acceptable since we assume that a server will not be idle if there are people in the queue. The set of states will be of the form

$$\mathcal{S} = \{(0, 0, 0), (0, 1, 0), (1, 0, 0)\} \cup \{(1, 1, k) : k = 0, 1, \dots, N - 2\}$$

The transition matrix will have the form

	000	010	100	110	111	112	...	1,1,N-3	1,1,N-2
000		$\lambda\alpha$	$\lambda(1-\alpha)$						
010	$\mu_2$			$\lambda$					
100	$\mu_1$			$\lambda$					
110		$\mu_1$	$\mu_2$		$\lambda$				
111						$\lambda$			
112					$\mu_1 + \mu_2$				
:									
1,1,N-3									$\lambda$
1,1,N-2							$\mu_1 + \mu_2$		

#### 4.1.2 Class QueueMM2dNState

Our characterization of each state fits nicely as a particular case of the `PropertiesState` class with three properties. Since we decided to work with numbered events rather than extending the `Event` class, we should implement the `SimpleMarkovClass`. In the following code you will see how we first model the State with the class `QueueMM2dNState` and then model the system implementing the class `QueueMM2dN`. These two class are placed in the same file `QueueMM2dN`, but they could be placed in separate files.

To model the State we begin by creating a constructor that assigns x, y, and z to the properties. We provide methods to access the three properties and a method to check whether the system is empty. We also implement the method `label` to override the one in the class `PropertiesState`.

#### 4.1.3 Class QueueMM2dN

There are two basic events that can occur: arrivals and service completions. We have to distinguish, however two types of service completions depending on whether the server that finishes is 1 or 2. Also, when the system is empty we have to distinguish between arrivals that go to server 1 and those that go to server 2. So in total we have five events which we number as follows

#### 4.1.4 Code

```

1 package examples.jmarkov;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7 import jmarkov.MarkovProcess;
8 import jmarkov.SimpleMarkovProcess;
9 import jmarkov.basic.Event;
10 import jmarkov.basic.EventsSet;
11 import jmarkov.basic.PropertiesState;
12 import jmarkov.basic.States;
13 import jmarkov.basic.StatesSet;
14
15 /**
16  * This class represents a system with 2 different exponential
17  * servers with rates mu1 and mu2, respectively, and arrival rate
18  * lambda.
19  * @author Germán Riaño. Universidad de los Andes.
20  */
21
22 public class QueueMM2dN extends SimpleMarkovProcess<MM2dNState, QMM2dNEvent> {
23     // Events
24     final int ARRIVAL = 0;
25     final int ARRIVAL1 = 1; // only for empty system
26     final int ARRIVAL2 = 2; // only for empty system
27     final int DEPARTURE1 = 3;
28     final int DEPARTURE2 = 4;
29     private double lambda;
30     private double mu1, mu2, alpha;
31     private int N;
32 }

```

```

33  /**
34  * Constructs a M/M/2d queue with arrival rate lambda and service
35  * rates mu1 and mu 2.
36  * @param lambda Arrival rate
37  * @param mu1 Server 1 rate
38  * @param mu2 Server 2 rate
39  * @param alpha Probability of an arriving customer choosing
40  * server 1 (if both idle)
41  * @param N Max number in the system
42  */
43  public QueueMM2dN(double lambda, double mu1, double mu2, double alpha, int N) {
44      super((new MM2dNState(0, 0, 0)), //
45            QMM2dNEvent.getAllEvents()); // num Events
46      this.lambda = lambda;
47      this.mu1 = mu1;
48      this.mu2 = mu2;
49      this.alpha = alpha;
50      this.N = N;
51  }
52
53  /**
54  * Returns an QueueMM2N object with arrival rate 4.0, service rate
55  * of the first server 2.0, service rate of the second server 3.0,
56  * probability of choose the first server 0.3 and capacity of 8
57  * customers in the system. Used by GUI
58  */
59  public QueueMM2dN() {
60      this(1.0, 2.0, 3.0, 0.3, 8);
61  }
62
63  /**
64  * Determines the active events
65  */
66  public @Override boolean active(MM2dNState i, QMM2dNEvent e) {
67      boolean result = false;
68      switch (e.getType()) {
69          case ARRIVAL:
70              result = ((i.getQSize() < N - 2) && (!i.isEmpty()));
71              break;
72          case ARRIVAL1:
73              result = i.isEmpty();
74              break;
75          case ARRIVAL2:
76              result = i.isEmpty();
77              break;
78
79          case DEPARTURE1:
80              result = (i.getStatus1() > 0);
81              break;
82          case DEPARTURE2:
83              result = (i.getStatus2() > 0);
84              break;
85      }
86      return result;
87  }
88
89  public @Override States<MM2dNState> dests(MM2dNState i, QMM2dNEvent e) {
90      int newx = i.getStatus1();
91      int newy = i.getStatus2();
92      int newz = i.getQSize();
93
94      switch (e.getType()) {
95          case ARRIVAL:
96              if (i.getStatus1() == 0) {
97                  newx = 1;
98              } // serv 1 desocupado
99              else if (i.getStatus2() == 0) {
100                  newy = 1;
101              } // serv 2 desocupado
102              else { // ambos ocupados
103                  newz = i.getQSize() + 1;
104              }
105              break;
106          case ARRIVAL1:
107              newx = 1;
108              break;
109          case ARRIVAL2:
110              newy = 1;
111              break;
112          case DEPARTURE1:
113              if (i.getQSize() != 0) {
114                  newx = 1;
115                  newz = i.getQSize() - 1;
116              } else {
117                  newx = 0;

```



```

118         }
119         break;
120     case DEPARTURE2:
121         if (i.getQSize() != 0) {
122             newy = 1;
123             newz = i.getQSize() - 1;
124         } else {
125             newy = 0;
126         }
127         break;
128     }
129     return new StatesSet<MM2dNState>( new MM2dNState(newx, newy, newz));
130 }
131
132 public @Override double rate(MM2dNState i, MM2dNState j, QMM2dNEvent e) {
133     double res = 0;
134     switch (e.getType()) {
135     case ARRIVAL:
136         res = lambda;
137         break;
138     case ARRIVAL1:
139         res = lambda * alpha;
140         break;
141     case ARRIVAL2:
142         res = lambda * (1 - alpha);
143         break;
144     case DEPARTURE1:
145         res = mu1;
146         break;
147     case DEPARTURE2:
148         res = mu2;
149         break;
150     }
151     return res;
152 }
153
154 @Override
155 public String description() {
156     return "M/M/2/N_SYSTEM\nQueueing_System_with_two_servers,_with_rates_"
157         + mu1 + "_and_" + mu2 + ".\nArrivals_are_Poisson_with_rate_"
158         + lambda + ",\nand_the_maximum_number_in_the_system_is_" + N;
159 }
160
161 /**
162  * This method just tests the class.
163  * @param a Not used
164  */
165 public static void main(String[] a) {
166     String stg;
167     BufferedReader rdr = new BufferedReader(
168         new InputStreamReader(System.in));
169     try {
170         System.out.println("Input_rate_");
171         stg = rdr.readLine();
172         double lda = Double.parseDouble(stg);
173         System.out.println("Service_rate_1_");
174         stg = rdr.readLine();
175         double mu1 = Double.parseDouble(stg);
176         System.out.println("Service_rate_2_");
177         stg = rdr.readLine();
178         double mu2 = Double.parseDouble(stg);
179         System.out.println("Provide_alpha_");
180         stg = rdr.readLine();
181         double alpha = Double.parseDouble(stg);
182         System.out.println("Max_in_the_system_");
183         stg = rdr.readLine();
184         int N = Integer.parseInt(stg);
185         QueueMM2dN theQueue = new QueueMM2dN(lda, mu1, mu2, alpha, N);
186         theQueue.showGUI();
187         theQueue.printAll();
188     } catch (IOException e) {
189     }
190 }
191 ;
192 }
193 } // class end
194
195 /**
196  * This is a particular case of propertiesState, with three
197  * properties, namely the server 1 and 2 status, plus the queue level.
198  * @author Germán Riaño. Universidad de los Andes.
199  */
200
201 class MM2dNState extends PropertiesState {
202

```

```

203
204 /**
205  * We identify each State with the triplet (x,y,z), where x and y
206  * are the status of the servers and z the number in queue (0,1,
207  * ..,N-2).
208  */
209
210 MM2dNState(int x, int y, int z) {
211     super(3); // Creates a PropertiesState with 3 properties.
212     this.prop[0] = x;
213     this.prop[1] = y;
214     this.prop[2] = z;
215 }
216
217 @Override
218 public void computeMOPs(MarkovProcess mp) {
219     setMOP(mp, "Status_Server_1", getStatus1());
220     setMOP(mp, "Status_Server_2", getStatus2());
221     setMOP(mp, "Queue_Length", getQSize());
222     setMOP(mp, "Number_in_System", getStatus1() + getStatus2() + getQSize());
223 }
224
225 /**
226  * Returns the status of the first Server
227  * @return Status of the first Server
228  */
229 public int getStatus1() {
230     return prop[0];
231 }
232
233 /**
234  * Returns the status of the second Server
235  * @return Status of the second Server
236  */
237 public int getStatus2() {
238     return prop[1];
239 }
240
241 /**
242  * Returns the size of the queue
243  * @return Status of the size of the queue
244  */
245 public int getQSize() {
246     return prop[2];
247 }
248
249 /**
250  * isEmpty detects is the system is empty. It comes handy when
251  * checking whether the events ARRIVAL1 and ARRIVAL2 are active.
252  */
253 boolean isEmpty() {
254     return (getStatus1() + getStatus2() + getQSize() == 0);
255 }
256
257 /**
258  * @see jmarkov.basic.State#isConsistent()
259  */
260 @Override
261 public boolean isConsistent() {
262     // TODO Complete
263     return true;
264 }
265
266 /**
267  * We implement label so that States are labeled 1, 1A, 1B, 2, 3,
268  * ..., N-2
269  */
270 @Override
271 public String label() {
272     String stg = "0";
273     if ((getStatus1() == 1) && (getStatus2() == 0))
274         stg = "1A";
275     if ((getStatus2() == 1) && (getStatus1() == 0))
276         stg = "1B";
277     if ((getStatus2() == 1) && (getStatus1() == 1))
278         stg = "2" + (2 + getQSize());
279     return stg;
280 }
281
282 /**
283  * This method gives a verbal description of the State.
284  */
285 @Override
286 public String description() {
287     String stg = "";

```

```

288     stg += "Server_1_is_" + ((getStatus1() == 1) ? "busy" : "idle");
289     stg += "_Server_2_is_" + ((getStatus2() == 1) ? "busy" : "idle");
290     stg += "_There_are_" + getQSize() + "_customers_waiting_in_queue.";
291     return stg;
292 }
293 }
294 }
295
296 class QMM2dNEvent extends Event {
297     /** Event types */
298     public enum Type {
299         /** An arrival */
300         ARRIVAL,
301         /** Arrival to server 1 (only for empty system) */
302         ARRIVAL1,
303         /** Arrival to server 2 (only for empty system) */
304         ARRIVAL2,
305         /** departure from server 1 */
306         DEPARTURE1,
307         /** departure from server 2 */
308         DEPARTURE2;
309     }
310
311     private Type type;
312
313     /**
314      * @param type
315      */
316     public QMM2dNEvent(Type type) {
317         super();
318         this.type = type;
319     }
320
321     /**
322      * @return Returns the type.
323      */
324     public final Type getType() {
325         return type;
326     }
327
328     /**
329      * @return the set of all events.
330      */
331     public static EventsSet<QMM2dNEvent> getAllEvents() {
332         EventsSet<QMM2dNEvent> evSet = new EventsSet<QMM2dNEvent>();
333         for (Type type : Type.values())
334             evSet.add(new QMM2dNEvent(type));
335         return evSet;
336     }
337 }
338
339 // Now we define main the class

```

## 4.2 Multiple Server Queue

In this example we generalize what we did in the previous example. Assume that a system has exponential arrivals with exponential arrivals. There are  $K$  distinct servers with service rates  $\mu_1, \mu_2, \dots, \mu_K$ . A customer that finds all servers busy joins a single FCFS queue, with capacity  $N - K$  (so there will be at most  $N$  customers in the system). A customer that finds all servers idle will choose among the idle servers according to relative intensities  $\alpha_k$ , i.e., he will choose server  $k$  with probability

$$\beta_k = \frac{\alpha_k}{\sum_{\ell \in \mathcal{I}} \alpha_\ell}, \quad k \in \mathcal{I}$$

where  $\mathcal{I}$  is the set of available servers.

### 4.2.1 The model

For this model we characterize each state by  $X(t) = (S(t), Q(t))$ , where  $S(t) = (S_1(t), \dots, S_K(t))$ , where  $S_k(t) = 1$  if  $k$ -th server is busy and 0 otherwise. The events that can occur are arrivals and departures. However we have to distinguish two type of arrivals. If there is no idle server the arriving customer joins the queue, and we will call this a non-directed arrival. Otherwise we call

it a directed arrival. We also make part of the event description the server where the arrival is directed. In order to represent this event we need a more sophisticated structure, so instead of just numbering the events we rather extend the class Event, creating an object with two integer fields (components): the type and the server. Then it is very easy to implement the functions `active`, `dest` and `rate` just by querying the values of the type and server associated with the state.

#### 4.2.2 Code

```

1 package examples.jmarkov;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7 import jmarkov.MarkovProcess;
8 import jmarkov.SimpleMarkovProcess;
9 import jmarkov.basic.Event;
10 import jmarkov.basic.EventsSet;
11 import jmarkov.basic.PropertiesState;
12 import jmarkov.basic.States;
13 import jmarkov.basic.StatesSet;
14
15 /**
16  * This class represents is a system with K different
17  * exponential servers with rates mu1, mu2, etc,
18  * respectively, and arrival rate lambda. A customer
19  * that finds more then one server idle chooses according
20  * to relative intensities
21  * <tex txt="$\alpha_1, \alpha_2, \ldots, \alpha_K$">
22  * alpha1, alpha2, etc</tex>. The probability of choosing
23  * idle server k will be given by
24  * <tex txt="\beta_k = \frac{\alpha_k}{\sum_{\ell \in \mathcal{I}} \alpha_{\ell}}\rangle,
25  * where $\mathcal{I}$ is the set of idle servers.">
26  * alpha(k) / sum( alpha(j)), where the sum is over the set of idle servers.
27  * </tex>
28  * @author Germán Riaño. Universidad de los Andes.
29  */
30 public class QueueMMKdN extends SimpleMarkovProcess<QueueMMKdNState, QueueMMKdNEvent> {
31     // Events
32
33     private double lambda;
34     private double[] mu, alpha;
35     private int K; // number of servers
36     private int N;
37     private static final int NDARRIVAL = QueueMMKdNEvent.NDARRIVAL;
38     private static final int DIRARRIVAL = QueueMMKdNEvent.DIRARRIVAL;
39     private static final int DEPARTURE = QueueMMKdNEvent.DEPARTURE;
40
41     /**
42      * Constructs a M/M/Kd queue with arrival rate lambda and service
43      * rates mu, relative probabilities of choosing each server alpha
44      * @param lambda Arrival rate
45      * @param mu Server rates
46      * @param alpha Relative probability of an arriving customer choosing each server.
47      * @param N Max number in the system
48      */
49     public QueueMMKdN(double lambda, double[] mu, double[] alpha, int N) {
50         super(
51             new QueueMMKdNState(mu.length, alpha),
52             QueueMMKdNEvent.getAllEvents(mu.length));
53         this.K = mu.length;
54         this.lambda = lambda;
55         this.mu = mu;
56         this.alpha = alpha;
57         this.N = N;
58     }
59
60     /**
61      * Returns an QueueMMKdN object with arrival rate 1.0,
62      * service rates of 2.0, 3.0 and 4.0;
63      * and capacity of 8 customers in the system.
64      * Used by GUI
65      */
66     public QueueMMKdN() {
67         this(1.0, new double[]{2,3,4}, new double[]{2,3,4}, 8);
68     }
69
70     /**
71      * Determines the active events.
72      */

```

```

73  @Override
74  public boolean active(QueueMMKdNState i, QueueMMKdNEvent e) {
75      boolean result = false;
76      switch (e.type) {
77          case (NDARRIVAL) : // NDARRIVAL occurs only if servers are busy and there is room in the
78                          result = (i.allBusy() && (i.getQSize() < N - K));
79                          break;
80          case (DIRARRIVAL) :
81              {
82                  result = (i.getStatus(e.server) == 0);
83                  //DirARRIVAL occurs if server is EMPTY.
84                  break;
85              }
86          case (DEPARTURE) :
87              { // ev.type == DEPARTURE
88                  result = (i.getStatus(e.server) == 1);
89                  //DEPARTURE occurs if server is busy.
90              }
91      }
92      return result;
93  }
94
95  /*
96   * Determines the possible destination event (actually one in this case).
97   */
98
99  @Override
100 public States<QueueMMKdNState> dests(QueueMMKdNState i, QueueMMKdNEvent e) {
101     int[] status = new int[K];
102     for (int k = 0; k < K; k++)
103         status[k] = i.getStatus(k); //copy current values
104     int Q = i.getQSize();
105     switch (e.type) {
106         case (NDARRIVAL) :
107             Q++; // non-directed ARRIVAL
108             break;
109         case (DIRARRIVAL) :
110             status[e.server] = 1; //directed ARRIVAL, picks a server.
111             break;
112         case (DEPARTURE) :
113             if (Q > 0) { //there is Queue
114                 status[e.server] = 1; //set (keeps) server busy
115                 Q--; // reduce queue
116             } else
117                 status[e.server] = 0; //set server idle
118     }
119     return new StatesSet<QueueMMKdNState>(new QueueMMKdNState(status, Q, alpha));
120 }
121
122 /*
123  * The rate is lambda, or mu for non-directed arrival and for departure.
124  * For directed arrival rate is lambda & prob(server is choosen)
125  * @see jmarkov.SimpleMarkovProcess#rate(jmarkov.State, jmarkov.State, jmarkov.Event)
126  */
127 @Override
128 public double rate(QueueMMKdNState i, QueueMMKdNState j, QueueMMKdNEvent e) {
129     double result = 0;
130
131     switch (e.type) {
132         case (DEPARTURE) :
133             result = mu[e.server];
134             break;
135         case (NDARRIVAL) :
136             result = lambda;
137             break; //non-directed arrival
138         case (DIRARRIVAL) :
139             result = i.prob(e.server) * lambda;
140     }
141     return result;
142 }
143
144 /**
145  * Main Method. This asks the user for parameters
146  * and tests the program.
147  * @param a Not used
148  */
149 public static void main(String[] a) {
150     BufferedReader rdr =
151         new BufferedReader(new InputStreamReader(System.in));
152     try {
153         System.out.println("Input_Rate:_");
154         double lda = Double.parseDouble(rdr.readLine());
155         System.out.println("Num_Servers:_");
156         int K = Integer.parseInt(rdr.readLine());
157         double mu[] = new double[K];

```

```

158         double alpha[] = new double[K];
159         for (int k = 0; k < K; k++) {
160             System.out.println("Service_rate,_server_" + (k + 1) + ":_");
161             mu[k] = Double.parseDouble(rdr.readLine());
162         }
163         for (int k = 0; k < K; k++) {
164             System.out.println(
165                 "Choosing_intensity,_server_" + (k + 1) + ":_");
166             alpha[k] = Double.parseDouble(rdr.readLine());
167         }
168         System.out.println("Max_in_system:_");
169         int N = Integer.parseInt(rdr.readLine());
170         QueueMMKdN theModel = new QueueMMKdN(la, mu, alpha, N);
171         theModel.showGUI();
172         //theModel.setDebugLevel(2);
173         theModel.printAll();
174     } catch (IOException e) {
175     };
176 }
177
178 /**
179  * @see jmarkov.SimpleMarkovProcess#description()
180  */
181 @Override
182 public String description() {
183     String stg = "M/M/k/N-SYSTEM\n\n";
184     stg += "Multiple_server_queue_with_" + this.K + "_different_servers\n";
185     stg += "Arrival_Rate=" + lambda + ",_Max_number_in_system_" + N;
186     return stg;
187 }
188
189 } //class end
190
191 /**
192  * This is a particular case of propertiesState, with K + 1
193  * properties, namely the server 1, 2, ..., K status, plus the queue level.
194  *
195  * @author Germán Riaño. Universidad de los Andes.
196  */
197 class QueueMMKdNState extends PropertiesState {
198     private int K; // number of servers
199     private double sumProb = -1; // sum of relative probabilities
200     private double[] alpha; //relative frequency of servers
201     private double[] beta; //probabilities for this state
202     /**
203      * Constructs a state for an empty system with K servers, and
204      * choosing intensities alpha.
205      * @param K Number of servers.
206      */
207     QueueMMKdNState(int K, double[] alpha) {
208         this(new int[K], 0, alpha);
209     }
210
211     /**
212      * We identify each State with a vector that counts the
213      * status for the k servers and
214      * the number in queue (0,1, ...,N-K).
215      */
216     QueueMMKdNState(int[] status, int Qsize, double[] alpha) {
217         super(alpha.length + 1);
218         this.K = alpha.length;
219         this.alpha = alpha;
220         this.beta = new double[K];
221         int sum = 0; // adds the number of busy server = people in service
222         for (int i = 0; i < K; i++) {
223             prop[i] = status[i];
224             sum += status[i];
225         }
226         prop[K] = Qsize;
227     }
228
229     /**
230      * Computes the MOPs
231      * @see jmarkov.basic.State#computeMOPs(MarkovProcess)
232      */
233     @Override
234     public void computeMOPs(MarkovProcess mp) {
235         double sum = 0.0;
236         for (int i = 0; i < K; i++) {
237             sum += getStatus(i);
238             setMOP(mp, "Server_Status_" + (i + 1), getStatus(i));
239         }
240         setMOP(mp, "Queue_Length", getQSize());
241         setMOP(mp, "Number_in_System", sum + getQSize());
242     }

```

```

243
244 /**
245  * Returns the status of the kth Server
246  * @param k server index
247  * @return Status of the kth Server
248  */
249 public int getStatus(int k) {
250     return prop[k];
251 }
252
253 /**
254  * Returns the size of the queue
255  * @return Status of the size of the queue
256  */
257 public int getQSize() {
258     return prop[K];
259 }
260
261 /**
262  * Determines if all servers are busy
263  * @return True, if all servers are busy. False, otherwise
264  */
265 public boolean allBusy() {
266     boolean result = true;
267     for (int k = 0; result && (k < K); k++)
268         result = result && (getStatus(k) == 1);
269     return result;
270 }
271
272 /**
273  * Determines if all servers are idle
274  * @return True, if all servers are idle. False, otherwise
275  */
276 public boolean allIdle() {
277     boolean result = true;
278     for (int k = 0; result && (k < K); k++)
279         result = result && (getStatus(k) == 0);
280     return result;
281 }
282
283 /**
284  * @see jmarkov.basic.State#isConsistent()
285  */
286 @Override
287 public boolean isConsistent() {
288     // TODO Complete
289     return true;
290 }
291
292 /**
293  * determines the sum of all intensities for idle servers. The result
294  * is kept in sumProb for future use.
295  */
296 private double sum() {
297     if (sumProb != -1)
298         return sumProb;
299     double res = 0;
300     for (int k = 0; k < K; k++) {
301         res += (1 - getStatus(k)) * alpha[k];
302     }
303     return (sumProb = res);
304 }
305
306 /**
307  * Detemines the probability of an idle server being choosen
308  * among idle servers. A customer that finds more than one server
309  * idle chooses according to relative intensities
310  * <tex txt="$\alpha_1, \alpha_2, \ldots, \alpha_K$">
311  * alpha1, alpha2, etc</tex>. The probability of choosing idle
312  * server k will be given by
313  * <tex txt="$\beta_k = \frac{\alpha_k}{\sum_{ell \in I} \alpha_{ell}}$">
314  * where $I$ is the set of idle servers.>
315  * alpha(k) / sum(j, alpha(j)), where the sum is over the set
316  * of idle servers. </tex>
317  * @param server index
318  * @return probability of an idle server being choosen
319  * among idle servers
320  */
321 public double prob(int server) {
322     if (beta[server] != 0)
323         return beta[server];
324     return (
325         beta[server] = (((1 - getStatus(server)) * alpha[server]) / sum()));
326 }
327
328 /**
329  * Returns a label with the format SxxQz, whre xx is the list of busy servers.
330  * @see jmarkov.basic.State#label()
331  */
332 @Override

```

```

328     public String label() {
329         String stg = "S";
330         for (int k = 0; k < K; k++) {
331             stg += (getStatus(k) == 1) ? " " + (k + 1) : " ";
332         }
333         return stg + "Q" + getQSize();
334     }
335
336     /*
337     * This method gives a verbal description of the State.
338     */
339     @Override
340     public String description() {
341         String stg = " ";
342         if (!allIdle())
343             stg += "Busy_Servers:";
344         else
345             stg += "No_one_in_service";
346         for (int k = 0; k < K; k++) {
347             stg += (getStatus(k) == 1) ? " " + (k + 1) + ", " : " ";
348         }
349         stg += "_There_are_" + getQSize() + "_customers_waiting_in_queue.";
350         return stg;
351     }
352 }
353 /**
354  *
355  * This class define the events.
356  * An event has two components: type which can have three values
357  * depending whether it represents a directed arrival, a
358  * non-directed arrival or a departure, and server, which
359  * represents the choosen server (if arrival) or the finishing
360  * server. For non-directed arrivals we set server -1 by convention.
361  *
362  *
363  * @author Germán Riaño
364  */
365
366 class QueueMMKdNEvent extends Event {
367     final static int NDARRIVAL = 0;
368     //Non directed arrival (when all servers are busy)
369     final static int DIRARRIVAL = 1; //Directed arrival chooses among server(s)
370     final static int DEPARTURE = 2;
371     int type; // ARRIVAL or DEPARTURE
372     /* server = chosen server if ARRIVAL finds many available,
373     * server = -1 if no server available
374     * server = finishing server if DEPARTURE event
375     */
376     int server;
377     QueueMMKdNEvent(int type, int server) {
378         this.type = type;
379         this.server = server;
380     }
381
382     static EventsSet<QueueMMKdNEvent> getAllEvents(int K) {
383         EventsSet<QueueMMKdNEvent> eSet = new EventsSet<QueueMMKdNEvent>();
384         eSet.add(new QueueMMKdNEvent(NDARRIVAL, -1));
385         for (int i = 0; i < K; i++) {
386             eSet.add(new QueueMMKdNEvent(DIRARRIVAL, i));
387         }
388         for (int i = 0; i < K; i++) {
389             eSet.add(new QueueMMKdNEvent(DEPARTURE, i));
390         }
391         return eSet;
392     }
393
394     /* (non-Javadoc)
395     * @see java.lang.Object#toString()
396     */
397     @Override
398     public String label() {
399         String stg = " ";
400         switch (type) {
401             case (NDARRIVAL) :
402                 stg += "Non-directed_arrival";
403                 break;
404             case (DIRARRIVAL) :
405                 stg += "Directed_arrival_to_server_" + (server + 1);
406                 break;
407             case (DEPARTURE) :
408                 stg += "Departure_from_server_" + (server + 1);
409                 break;
410         }
411         return stg;
412     }

```



```

413 } //end class
414 // Lets start defining the State
415
416
417 // Now we define the main class

```

## 4.3 Drive Thru

### 4.3.1 Code

```

1  package examples.jmarkov;
2
3  import static examples.jmarkov.DriveThruEvent.Type.ARRIVAL;
4  import static examples.jmarkov.DriveThruEvent.Type.MIC.COMPLETION;
5  import static examples.jmarkov.DriveThruEvent.Type.SERVICE.COMPLETION;
6  import static examples.jmarkov.DriveThruState.CustStatus.BLOCKED_DONE;
7  import static examples.jmarkov.DriveThruState.CustStatus.COOKING;
8  import static examples.jmarkov.DriveThruState.CustStatus.EMPTY;
9  import static examples.jmarkov.DriveThruState.CustStatus.ORDERING;
10 import static examples.jmarkov.DriveThruState.CustStatus.WAIT_MIC;
11
12 import java.io.PrintWriter;
13
14 import jmarkov.MarkovProcess;
15 import jmarkov.SimpleMarkovProcess;
16 import jmarkov.basic.Event;
17 import jmarkov.basic.EventsSet;
18 import jmarkov.basic.State;
19 import jmarkov.basic.States;
20 import jmarkov.basic.StatesSet;
21 import jmarkov.basic.exceptions.NotUnichainException;
22 import examples.jmarkov.DriveThruState.CustStatus;
23
24 /**
25  * This class implements a Drive Thru. Extends
26  * SimpleMarkovProcess.
27  *
28  * @author Margarita Arana y Gloria Díaz. Universidad de los Andes.
29  * Mod: Germán Riaño (2004)
30  * @version 1.0a
31  */
32 public class DriveThru extends
33     SimpleMarkovProcess<DriveThruState, DriveThruEvent> {
34
35     double lambda; // arrival rate
36     double mu1; // Service rate for server 1
37     double mu2; // Service rate for server 2
38     int M; // Maximum number of clients in the system
39     int S; // Number of servers
40     int N; // Number of places between the window and the microphone
41
42     /**
43      * Constructor de un DriveThru.
44      *
45      * @param lambda
46      *         Tasa de arribos
47      * @param mu1
48      *         Tasa de servicios del micrófono
49      * @param mu2
50      *         Tasa de servicios de la ventana
51      * @param M
52      *         Número máximo de entidades en el sistema
53      * @param S
54      *         Número de servidores
55      * @param N
56      *         Número de puestos entre la ventana y el micrófono
57      */
58     public DriveThru(double lambda, double mu1, double mu2, int M, int S, int N) {
59         super((new DriveThruState(N, S)), DriveThruEvent.getAllEvents(N));
60         this.lambda = lambda;
61         this.mu1 = mu1;
62         this.mu2 = mu2;
63         this.M = M;
64         this.S = S;
65         this.N = N;
66     }
67
68     /**
69      * Default constructor for GUI.
70      */
71

```

```

72 public DriveThru() {
73     this(80.0, 12.0, 30.0, 4, 2, 1);
74 }
75
76 /**
77  * Determines when the states are active for each state.
78  *
79  * @see SimpleMarkovProcess#active(State, Event)
80  */
81
82 @Override
83 public boolean active(DriveThruState s, DriveThruEvent ev) {
84     boolean result = false;
85     switch (ev.getType()) {
86     case ARRIVAL:
87         // un carro puede llegar si hay espacio en cola
88         result = (s.getQLength() < M - N - 1);
89         break;
90     case MIC.COMPLETION:
91         // se puede terminar de tomar la orden si una persona esta haciendo
92         // el pedido
93         result = (s.getMicStatus() == ORDERING);
94         break;
95     default:
96         // se puede terminar una orden si la persona correspondiente la esta
97         // esperando
98         if (ev.getPos() == N) {
99             result = (s.getMicStatus() == COOKING);
100         } else {
101             result = (s.getStatus(ev.getPos()) == COOKING);
102         }
103     }
104     return result;
105 }
106
107 /**
108  * Computes the rate: the rate is lambda if an arrival occurs,
109  * the rate is mul if a service type one is finished,
110  * the rate is mu2 if a service type two is finished.
111  *
112  * @see SimpleMarkovProcess#rate(State, State, Event)
113  */
114 @Override
115 public double rate(DriveThruState i, DriveThruState j, DriveThruEvent e) {
116     switch (e.getType()) {
117     case ARRIVAL:
118         return lambda;
119     case MIC.COMPLETION:
120         return mul;
121     default:
122         return mu2;
123     }
124 }
125
126 /**
127  * Computes the status of the destination when an event occurs
128  *
129  * @see SimpleMarkovProcess#dests(State, Event)
130  */
131
132 @Override
133 public States<DriveThruState> dests(DriveThruState i, DriveThruEvent e) {
134     int numServ = i.getAvlServs();
135     CustStatus[] status = i.getStatus();
136     CustStatus newMic = i.getMicStatus();
137     int newQsize = i.getQLength();
138     int numGone = 0;
139     boolean micMoves = false;
140     int k; // utility counter
141
142     switch (e.getType()) {
143     case ARRIVAL:
144         if (i.getMicStatus() == EMPTY && numServ > 0) {
145             newMic = ORDERING;
146             numServ = numServ - 1;
147         } else if (i.getMicStatus() == EMPTY && numServ == 0) {
148             newMic = WAIT_MIC;
149         } else if (i.getQLength() < M - N - 1) {
150             newQsize = i.getQLength() + 1;
151         }
152         break;
153     }
154 }
155
156

```

```

157     case MIC.COMPLETION:
158         newMic = COOKING;
159         for (k = 0; ((k < N) && (status[k] != EMPTY)); k++)
160             ;
161
162         if (k != N) {
163             status[k] = COOKING;
164             newMic = EMPTY;
165             micMoves = true;
166         }
167         break;
168
169     default:
170         numServ = numServ + 1;
171         int p = e.getPos();
172         if (p > 0 && p < N) {
173
174             status[p] = BLOCKED.DONE;
175         } else if (p == N) {
176             newMic = BLOCKED.DONE;
177         } else {
178
179             status[0] = EMPTY;
180
181             int pos1, pos2;
182
183             for (k = 1; ((k < N) && status[k] == BLOCKED.DONE); k++)
184                 ;
185             numGone = k;
186             if (k != N) {
187                 pos1 = k;
188                 pos2 = N - 1;
189                 for (k = pos1; k <= pos2; k++) {
190                     status[k - numGone] = status[k];
191                 }
192             }
193             for (k = N - numGone; k < N; k++) {
194                 status[k] = EMPTY;
195             }
196             if (newMic == COOKING) {
197                 status[N - numGone] = newMic;
198                 newMic = EMPTY;
199                 micMoves = true;
200             } else if (newMic == BLOCKED.DONE) {
201                 newMic = EMPTY;
202                 micMoves = true;
203             }
204         }
205         break;
206     } // end switch
207
208     if (newMic == WAIT_MIC && numServ > 0) {
209         newMic = ORDERING;
210         numServ--;
211     }
212     if (micMoves) {
213         if (i.getQLength() > 0 && numServ > 0) {
214             newMic = ORDERING;
215             numServ = numServ - 1;
216             newQsize = i.getQLength() - 1;
217         } else if (i.getQLength() > 0 && numServ == 0) {
218             newMic = WAIT_MIC;
219             newQsize = i.getQLength() - 1;
220         }
221     }
222     StatesSet<DriveThruState> set = new StatesSet<DriveThruState>();
223     set.add(new DriveThruState(status, newMic, newQsize, numServ));
224     return set;
225 } // end dests
226
227 @Override
228 public String description() {
229     return "SISTEMA_DRIVE_THRU." + "\nTasa_de_Entrada" + lambda
230         + "\nTasa_en_el_Mic" + mu1 + "\nTasa_de_servicio"
231         + mu2 + "\nPosici n del mic" + N + "\nServidores"
232         + S + "\nCapacidad del sistema" + M;
233 }
234
235 /**
236  * Print all waiting times associated with each MOP
237  */
238 @Override
239 public int printMOPs(PrintWriter out, int width, int decimals) {
240     int namesWidth = super.printMOPs(out, width, decimals);
241     // this rate work for all MOPs

```

```

242     double ldaEff;
243     try {
244         ldaEff = getEventRate(ARRIVAL.ordinal());
245         String[] names = getMOPNames();
246         double waitTime;
247         int N = names.length;
248         namesWidth += 20;
249         for (int i = 0; i < N; i++) {
250             waitTime = 60 * getMOPsAvg(names[i]) / ldaEff;
251             String name = "Waiting_time_for_" + names[i];
252             out.println(pad(name, namesWidth, false)
253                         + pad(waitTime, width, decimals) + "_minutes");
254         }
255     } catch (NotUnichainException e) {
256         out.println(e);
257     }
258     return namesWidth;
259 }
260
261 /**
262  * Main method.
263  *
264  * @param a
265  *         Not used.
266  */
267 public static void main(String[] a) {
268     // as in handout:
269     DriveThru theDT = new DriveThru(80.0, 12.0, 30.0, 4, 2, 1);
270     // DriveThru theDT = new DriveThru(80.0, 120.0, 30.0, 4, 2, 2);
271     theDT.setDebugLevel(5);
272
273     theDT.showGUI();
274     theDT.printAll();
275     theDT.printMOPs();
276 }
277
278 } // class end
279
280 /**
281  * This is a particular case of PropertiesState. Here, N is the position of the
282  * microphone. The first N-1 components represent the status of the first queue, the
283  * component N is the status of the microphone, the component N+1 is the number of clients in
284  * the queue, and N+2 are the available servers.
285  */
286 class DriveThruState extends State {
287
288     // private int micPos;
289     // private CustStatus micStatus;
290     private int numQ;
291     private int avlServ;
292     private CustStatus[] prop = null;
293
294     /**
295      * This enumeration shows the different status for a customer.
296      *
297      */
298     public enum CustStatus {
299         /** Empty space. */
300         EMPTY,
301         /** In service. */
302         ORDERING,
303         /** A client in the microphone, but there are no servers available. */
304         WAIT_MIC,
305         /** The client order is being prepared. */
306         COOKING,
307         /** The order is ready but the client is blocked. */
308         BLOCKED_DONE;
309     }
310
311     /**
312      * Builds a State representing an empty system
313      *
314      * @param micPos
315      * @param serv
316      */
317     DriveThruState(int micPos, int serv) {
318         this(new CustStatus[micPos], EMPTY, 0, serv);
319         for (int i = 0; i < prop.length; i++) {
320             prop[i] = EMPTY;
321         }
322     }
323
324     /**
325      * Builds a DriveThru state.
326      *

```

```

327 * @param vec
328 * The states from the window until the microphone,
329 * without including the microphone.
330 * @param mic
331 * Microphone status.
332 * @param numQ
333 * Number of clients in the queue.
334 * @param avServs
335 * Number of servers available.
336 */
337
338 DriveThruState(CustStatus[] statusVec, CustStatus micStatus, int numQ,
339               int avServs) {
340     prop = new CustStatus[statusVec.length + 1];
341     int micPos = statusVec.length;
342     System.arraycopy(statusVec, 0, prop, 0, micPos);
343     prop[micPos] = micStatus;
344     this.numQ = numQ;
345     this.avlServ = avServs;
346 }
347
348 /**
349 * Compute all the MOPs for this state
350 */
351 @Override
352 public void computeMOPs(MarkovProcess mp) {
353     int servEtapa1 = 0;
354     int servEtapa2 = 0;
355     int blockedDone = 0;
356     int blockedBefore = 0;
357     int total = 0;
358     for (CustStatus s : prop) {
359         servEtapa1 += (s == ORDERING) ? 1 : 0;
360         servEtapa2 += (s == COOKING) ? 1 : 0;
361         blockedDone += (s == BLOCKED_DONE) ? 1 : 0;
362         blockedBefore += (s == WAIT_MIC) ? 1 : 0;
363         total += (s != EMPTY) ? 1 : 0;
364     }
365     setMOP(mp, "Tamano_Cola", getQLength());
366     setMOP(mp, "Serv_Ocupados_Microfono_", servEtapa1);
367     setMOP(mp, "Serv_Ocupados_Cocinando", servEtapa2);
368     setMOP(mp, "Serv_Ocupados_", servEtapa1 + servEtapa2);
369     setMOP(mp, "Clientes_Bloqueados_antes_de_ordenar", blockedBefore);
370     setMOP(mp, "Clientes_Bloqueados_con_orden_lista", blockedDone);
371     setMOP(mp, "Clientes_Bloqueados", blockedBefore + blockedDone);
372     setMOP(mp, "Total_clientes_en_Espera", blockedBefore + blockedDone
373           + getQLength());
374     setMOP(mp, "Total_Clientes_", total + getQLength());
375 }
376
377 /**
378 * Get the number of clients in the queue.
379 *
380 * @return Number of clients in the queue.
381 */
382 public int getQLength() {
383     return numQ;
384 }
385
386 /**
387 * Get the status of the of the i-th component.
388 *
389 * @param i
390 * index of the component
391 *
392 * @return Status of the i-th component.
393 */
394 public CustStatus getStatus(int i) {
395     return prop[i];
396 }
397
398 /**
399 * Get the vector of clients statuses.
400 *
401 * @return Status of components 0 to N-1.
402 */
403 public CustStatus[] getStatus() {
404     int micPos = getMicPos();
405     CustStatus[] status = new CustStatus[micPos];
406     System.arraycopy(prop, 0, status, 0, micPos);
407     return status;
408 }
409
410 /**
411 * Get the status of the window.

```

```

412     *
413     * @return The status of the client at the microphone.
414     */
415     public CustStatus getMicStatus() {
416         int n = prop.length - 1;
417         return prop[n];
418     }
419
420     /**
421     * Return the mic position.
422     *
423     * @return mic position index
424     */
425     public int getMicPos() {
426         return prop.length - 1;
427     }
428
429     /**
430     * Get the status of the window
431     *
432     * @return Status of the window.
433     */
434     public CustStatus getVentana() {
435         return prop[0];
436     }
437
438     /**
439     * Computes the number of available servers.
440     *
441     * @return Number of available servers.
442     */
443     public int getAvlServs() {
444         return avlServ;
445     }
446
447     /**
448     * @see jmarkov.basic.State#isConsistent()
449     */
450     @Override
451     public boolean isConsistent() {
452         // TODO Complete
453         return true;
454     }
455
456     @Override
457     public String label() {
458         String stg = "";
459         for (CustStatus s : prop) {
460             switch (s) {
461                 case EMPTY:
462                     stg += "0";
463                     break;
464                 case ORDERING:
465                     stg += "m";
466                     break;
467                 case WAIT_MIC:
468                     stg += "w";
469                     break;
470                 case COOKING:
471                     stg += "c";
472                     break;
473                 case BLOCKED_DONE:
474                     stg += "b";
475                     break;
476             }
477         }
478         return stg + "Q" + numQ;
479         // return stg + "Q" + prop[micPos + 1] + "S" + prop[micPos + 2];
480     }
481
482     String statusDesc(CustStatus stat) {
483         switch (stat) {
484             case EMPTY:
485                 return "empty";
486             case ORDERING:
487                 return "ordering,";
488             case WAIT_MIC:
489                 return "waiting";
490             case COOKING:
491                 return "cooking";
492             default: // DONE
493                 return "blocked";
494         }
495     }
496

```

```

497  /**
498  * Describes the State
499  *
500  * @see jmarkov.basic.State#description()
501  */
502  @Override
503  public String description() {
504      String stg = "";
505      int N = getMicPos();
506      stg = "Queue_CustStatus:_" +
507      for (int i = 0; i < N; i++) {
508          stg += statusDesc(getStatus(i));
509          stg += (i < N - 1) ? ",_" : "";
510      }
511      stg += ")._Mic_status:_" + statusDesc(getMicStatus());
512      stg += "._Queue_Size:_" + getQLength();
513      return stg;
514  }
515
516  /**
517  * @see jmarkov.basic.State#compareTo(jmarkov.basic.State)
518  */
519  @Override
520  public int compareTo(State j) {
521      if (!(j instanceof DriveThruState))
522          throw new IllegalArgumentException("Comparing_wrong_types!");
523      DriveThruState u = (DriveThruState) j;
524      int micPos = getMicPos();
525      for (int k = 0; k <= micPos; k++) {
526          if (getStatus(k).ordinal() > u.getStatus(k).ordinal())
527              return +1;
528          if (getStatus(k).ordinal() < u.getStatus(k).ordinal())
529              return -1;
530      }
531      if (getQLength() > u.getQLength())
532          return +1;
533      if (getQLength() < u.getQLength())
534          return -1;
535      if (getAvlSrvs() > u.getAvlSrvs())
536          return +1;
537      if (getAvlSrvs() < u.getAvlSrvs())
538          return -1;
539      return 0;
540  }
541  }
542  }
543
544  /**
545  * This class implements the events in a Drive Thru.
546  */
547  class DriveThruEvent extends jmarkov.basic.Event {
548      /** Event types. */
549      public static enum Type {
550          /** Arrivale to the system. */
551          ARRIVAL,
552          /** Car at mic finishes service. */
553          MIC.COMPLETION,
554          /** Service completion for somebody who ordered. */
555          SERVICE.COMPLETION;
556      }
557
558      private Type type; // event type
559      private int position; // Position of the client whose order is complete
560
561      /**
562       * Creates an ARRIVAL or MIC.COMPLETION event.
563       *
564       * @param type
565       */
566      public DriveThruEvent(Type type) {
567          assert (type == ARRIVAL || type == MIC.COMPLETION);
568          this.type = type;
569      }
570
571      /**
572       * Creates a Service Completion event at he given position.
573       *
574       * @param position
575       *     Postion where the event occurs ( 0-based ).
576       */
577      public DriveThruEvent(int position) {
578          this.type = SERVICE.COMPLETION;
579          this.position = position;
580      }
581  }

```

```

582  /**
583   * @return position where this event occurs. (valid only if type ==
584   *      SERVICE_COMPLETION).
585   */
586  public int getPos() {
587      assert (type == SERVICE_COMPLETION);
588      return position;
589  }
590
591  /**
592   * @return event type
593   */
594  public Type getType() {
595      return type;
596  }
597
598  /**
599   * @param micPos
600   * @return A set with all the events in the system.
601   */
602  public static EventsSet<DriveThruEvent> getAllEvents(int micPos) {
603      EventsSet<DriveThruEvent> eSet = new EventsSet<DriveThruEvent>();
604      eSet.add(new DriveThruEvent(ARRIVAL));
605      eSet.add(new DriveThruEvent(MIC_COMPLETION));
606      for (int i = 0; i <= micPos; i++)
607          eSet.add(new DriveThruEvent(i));
608      return eSet;
609  }
610
611  @Override
612  public String label() {
613      String stg = "";
614      switch (type) {
615          case ARRIVAL:
616              stg = "Arrival";
617              break;
618          case MIC_COMPLETION:
619              stg = "MicEnd";
620              break;
621          default:
622              stg = "SrvEnd(" + position + ")";
623      }
624      return stg;
625  }
626
627 }

```

## 4.3.2 Results

### Output for Drive Thru

```

1  SISTEMA DRIVE THRU.
2  Tasa de Entrada   = 80.0
3  Tasa en el Mic    = 120.0
4  Tasa de servicio 2 = 30.0
5  Posici{\ 'o}n del mic = 5
6  Servidores ===== 4
7  Cap_en_el_sistema = 14
8
9
10
11  System_has_498_States.
12
13
14  MEASURES_OF_PERFORMANCE
15
16  NAME_____MEAN_____SDEV
17  Tamano_Cola_____4.503_____2.693
18  Serv_Ocupados_Microfono_____0.550_____0.498
19  Serv_Ocupados_Cocinando_____2.199_____1.165
20  Serv_Ocupados_____2.749_____1.088
21  Clientes_Bloqueados_antes_de_ordenar_____0.112_____0.316
22  Clientes_Bloqueados_con_orden_lista_____1.540_____1.646
23  Clientes_Bloqueados_____1.652_____1.604
24  Total_clientes_en_Espera_____6.155_____3.487
25  Total_Clientes_____8.903_____3.396
26
27  EVENTS_OCCURANCE_RATES
28  NAME_____MEAN_RATE
29  Arrival_____65.965
30  MicEnd_____65.965

```



```

31 SrvEnd(0) .....28.019
32 SrvEnd(1) .....9.927
33 SrvEnd(2) .....9.446
34 SrvEnd(3) .....8.333
35 SrvEnd(4) .....6.114
36 SrvEnd(5) .....4.126
37
38 Tiempo_de_espera_para_Tamano_Cola: 4.096 minutos
39 Tiempo_de_espera_para_Serv_Ocupados_Microfono: 0.5 minutos
40 Tiempo_de_espera_para_Serv_Ocupados_Cocinando: 2 minutos
41 Tiempo_de_espera_para_Serv_Ocupados: 2.5 minutos
42 Tiempo_de_espera_para_Clientes_Bloqueados_antes_de_ordenar: 0.102 minutos
43 Tiempo_de_espera_para_Clientes_Bloqueados_con_orden_lista: 1.4 minutos
44 Tiempo_de_espera_para_Clientes_Bloqueados: 1.503 minutos
45 Tiempo_de_espera_para_Total_clientes_en_Espera: 5.598 minutos
46 Tiempo_de_espera_para_Total_Clientes: 8.098 minutos

```

## 5 Modeling Quasi-Birth and Death Processes

In this section we give a brief description of Quasi-Birth and Death Processes (QBD), and explain how they can be modeled using jMarkov. QBD are Markov Processes with an infinite space state, but with a very specific repetitive structure that makes them quite tractable.

### 5.1 Quasi-Birth and Death Processes

Consider a Markov process  $\{X(t) : t \geq 0\}$  with a two dimensional state space  $\mathcal{S} = \{(n, i) : n \geq 0, 0 \leq i \leq m\}$ . The first coordinate  $n$  is called the *level* of the process and the second coordinate  $i$  is called the *phase*. We assume that the number of phases  $m$  is finite. In applications, the level usually represents the number of items in the system, whereas the phase might represent different stages of a service process.

We will assume that, in one step transition, this process can go only to the states in the same level or to adjacent levels. This characteristic is analogous to a Birth and Death Process, where the only allowed transitions are to the two adjacent states (see, e.g [5]). Transitions can be from state  $(n, i)$  to state  $(n', i')$  only if  $n' = n$ ,  $n' = n - 1$  or  $n' = n + 1$ , and, for  $n \geq 1$  the transition rate is independent of the level  $n$ . Therefore, the generator matrix,  $\mathbf{Q}$ , has the following structure

$$\mathbf{Q} = \begin{bmatrix} \mathbf{B}_{00} & \mathbf{B}_{01} & & & \\ \mathbf{B}_{10} & \mathbf{A}_1 & \mathbf{A}_0 & & \\ & \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{A}_0 & \\ & & \ddots & \ddots & \ddots \end{bmatrix},$$

where, as usual, the rows add up to 0. An infinite Markov Process with the conditions described above is called a Quasi-Birth and Death Process (QBD).

In general, the level zero might have a number of phases  $m_0 \neq m$ . We will call these first  $m_0$  states the *boundary states*, and all other states will be called *typical states*. Note that matrix  $\mathbf{B}_{00}$  has size  $m_0 \times m_0$ , whereas  $\mathbf{B}_{01}$  and  $\mathbf{B}_{10}$  are matrices of sizes  $(m_0 \times m)$  and  $(m \times m_0)$ , respectively. Assume that the QBD is an ergodic Markov Chain. As a result, there is a steady state distribution  $\boldsymbol{\pi}$  that is the unique solution  $\boldsymbol{\pi}$  to the system  $\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$ ,  $\boldsymbol{\pi}\mathbf{1} = 1$ . Divide this  $\boldsymbol{\pi}$  vector by levels, analogously to the way  $\mathbf{Q}$  was divided, as

$$\boldsymbol{\pi} = [\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \dots].$$

Then, it can be shown that a solution exist that satisfy

$$\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \mathbf{R}, \quad n > 1,$$

where  $\mathbf{R}$  is a constant square matrix of order  $m$  [7]. This  $\mathbf{R}$  is the solution to the equation

$$\mathbf{A}_0 + \mathbf{R}\mathbf{A}_1 + \mathbf{R}^2\mathbf{A}_2 = \mathbf{0}.$$

There are various algorithms that can be used to compute the matrix  $\mathbf{R}$ . For example, you can start with any initial guess  $\mathbf{R}_0$  and obtain a series of  $\mathbf{R}_k$  through iterations of the form

$$\mathbf{R}_{k+1} = -(\mathbf{A}_0 + \mathbf{R}_k^2 \mathbf{A}_2) \mathbf{A}_1^{-1}.$$

This process is shown to converge (and  $\mathbf{A}_1$  does have an inverse). More elaborated algorithms are presented in Latouche and Ramaswami [6]. Once  $\mathbf{R}$  has been determined then  $\boldsymbol{\pi}_0$  and  $\boldsymbol{\pi}_1$  are determined by solving the following linear system of equations

$$\begin{bmatrix} \boldsymbol{\pi}_0 & \boldsymbol{\pi}_1 \end{bmatrix} \begin{bmatrix} \mathbf{B}_{00} & \mathbf{B}_{01} \\ \mathbf{B}_{10} & \mathbf{A}_1 + \mathbf{R} \mathbf{A}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \end{bmatrix}$$

$$\boldsymbol{\pi}_0 \mathbf{1} + \boldsymbol{\pi}_1 (\mathbf{I} - \mathbf{R})^{-1} \mathbf{1} = 1.$$

## 5.2 Measures of performance for QBDs

We consider two types of measures of performance that can be defined in a QBD model. The first type can be seen as a reward  $r_i$  received whenever the system is in phase  $i$ , independent of the level, for level  $n \geq 1$ . The long-run value for such a measure of performance is computed according to

$$\sum_{n=1}^{\infty} \boldsymbol{\pi}_n \mathbf{r} = \boldsymbol{\pi}_1 (\mathbf{I} - \mathbf{R})^{-1} \mathbf{r},$$

where  $\mathbf{r}$  is an  $m$ -size column vector with components  $r_i$ . The second type of reward has the form  $nr_i$ , whenever the system is in phase  $i$  of level  $n$ . Its long-run value is

$$\sum_{n=1}^{\infty} n \boldsymbol{\pi}_n \mathbf{r} = \boldsymbol{\pi}_1 \mathbf{R} (\mathbf{I} - \mathbf{R})^{-2} \mathbf{r}.$$

## 5.3 Modeling QBD with jQBD

Modeling QBD with jMarkov is similar to modeling a Markov Processes. Again, the user has to code the states, the events, and then define the dynamics of the system through **active**, **dests**, and **rate**. The main difference is that special care needs to be taken when defining the destination states for the typical states. Rather than defining a new level for the destination state, the user should give a new *relative* level, which can be -1, 0, or +1. This is accomplished by using two different classes to define states. The current state of the system is a **GeomState**, but the destination states are **GeomRelState**. The process itself must extend the class **GeomProcess**, which in turn is an extension of **MarkovProcess**.

The building algorithm uses the information stored about the dynamics of the process to explore the graph and build only the first three levels of the system. From this, it is straightforward to extract matrices  $\mathbf{B}_{00}$ ,  $\mathbf{B}_{01}$ ,  $\mathbf{B}_{10}$ ,  $\mathbf{A}_0$ ,  $\mathbf{A}_1$ , and  $\mathbf{A}_2$ . Once these matrices are obtained, the stability condition is checked. If the system is found to be stable, then the matrices  $\mathbf{A}_0$ ,  $\mathbf{A}_1$ , and  $\mathbf{A}_2$  are passed to the solver, which takes care of computing the matrix  $\mathbf{R}$  and the steady state probabilities vectors  $\boldsymbol{\pi}_0$  and  $\boldsymbol{\pi}_1$ , using the formulas described above. The implemented solver (**MtjLogRedSolver**) uses the logarithmic reduction algorithm [6]. This class uses MTJ for matrices manipulations. There are also mechanisms to define both types of measures of performance mentioned above, and jQBD can compute the long run average value for all of them.

## 5.4 An Example

To illustrate the modeling process with jQBD, we will show the previous steps with a simple example. Consider a infinite queue with a station that has a single hyper-exponential server with  $n$  service phases, with probability  $\alpha_i$  to reach the service phase  $i$  and with service rate  $\mu_i$  at phase  $i$ , where  $0 \leq i \leq n$ . The station is fed from an external source according to a Poisson processes with rate  $\lambda$ . We will use this model as an illustrative example of a QBD process, and will show how each of the previous steps is performed for this example. Of course all measures of performance for this system can be readily obtained in closed form since it is a particular case of an  $M/G/1$ , but we chose this example because of its simplicity. The code below actually models any general phase-type distribution, so the hyper-geometric will be a particular case.

- **States:** Because of the memoryless property, the state of the system is fully characterized by an integer valued vector  $\mathbf{x} = (x_1, x_2)$ , where  $x_1 \geq 0$  represents the number of items in the system and  $0 \leq x_2 \leq n$  represents the current phase of the service process. Note that, knowing this, we can know how many items are in service and how many are queuing. It is important to highlight that the computational representation uses only the phase of the system ( $x_2$ ) because the level ( $x_1$ ) is managed internally by the framework.
- **Events:** An event occurs whenever an item arrives to the system or finishes processing at a particular service phase  $0 \leq i \leq n$ . Therefore, we will define the set of possible events as  $\mathcal{E} = \{a, c_1, c_2, \dots, c_n\}$ , where the event  $a$  represents an arrival to the system and an event  $c_i$  represents the completion of a service in phase  $i$ .
- **Markov Process:** We elected to implement `GeomProcess`, which implied coding the following three methods:
  - **active (i,e):** Since the queue is an infinite QBD process the event  $a$  is always active, and the events  $c_i, 0 \leq i \leq n$  are active if there is an item at workstation on service phase  $i$ . The code to achieve this can be seen in Figure 6.
  - **dests (i,e,j):** When the event  $a$  occurs there is always an increment on the system level, but you need to consider if the server is idle or busy. When the server is idle the new customer could start in any of the  $n$  service phases, then the system could reach anyone of the first level  $n$  states with probability  $\alpha_i$ . On the other hand, if the server is busy on service phase  $i$ , the system will reach the next level state with the same service phase  $i$ .  
On the other hand, when the server finishes one service  $c_i$ , no matter which phase type, the level of the system is reduced by one, but you need to consider if the system is in level 1 or if it is in level 2 or above. When the level is 1, the system reach the unique state  $(0,0)$  where there are no customer in the system and the server is idle. On the other hand, if the system level is equal or greater than 2, the system could reach any of the  $n$  states in the level below with probability  $\alpha_i$ . The Java code can be seen in Figure 7.
  - **rate (i,e):** The rate of occurrence of event  $a$  is given simply by  $\lambda$  and the rate of occurrence of an event  $c_i$  is given by  $\mu_i$ . In Figure 8 you can see the corresponding code.
- **MOPs:** Using the MOPS types defined in jQBD component, we will illustrate its use calculating the expected WIP on the system.

Finally, the output obtained after running the model can be seen in the Graphical User Interface (GUI) in Figure 9. There is no need to use the GUI, but it is helpful to do so during the first stages of development, to make sure that all transitions are being generated as expected. All the measures of performance defined can be extracted by convenience methods defined in the API or a report printed to standard output. Such a report can be seen in Figure 10.

```

1  public int getCurPH() {
2      if (type == ARRIVAL)
3          throw new IllegalArgumentException(
4              "Current phase is not defined for event " + ARRIVAL);
5      return curPH;
6  }
7
8  /**
9   * @return Returns the type.
10   */
11  public Type getType() {
12      return type;
13  }

```

Figure 6: Active method of class HiperExQueue.java

```

1      // finish in phase n
2      E.add(new HiperExQueueEvent(FINISH_SERVICE, n));
3  }
4  return E;
5  }
6
7  @Override
8  public String label() {
9      String stg = "";
10     switch (type) {
11     case ARRIVAL:
12         stg = "Arrival";
13         break;
14     case FINISH_SERVICE:
15         stg = "Ph(" + curPH + ")";
16     }
17     return stg;
18 }
19 }
20
21 /**
22  * * This class define the states in the queue.
23  * @author Julio Goez – German Riano. Universidad de los Andes.
24  */
25 class HiperExQueueState extends PropertiesState {
26
27     /**
28     * We identify the states with the curPH of server in station, (1,
29     * ..,n) or 0 if idle.

```

Figure 7: dests method of class HiperExQueue.java

## 6 Modeling Priority Queues: incorporating phase-type distributions with jPhase

In this section we introduce an example to illustrate the use of jMarkov, particularly the jQBD and jPhase modules. We do not aim to describe the implementation in full here, which is available at [4], but to highlight some of the key steps in modeling with jMarkov.

We consider a first-come-first-serve queue with a single server and two classes of jobs that receive service, one with high priority and the other with low priority. We also refer to high and low priority jobs as being of class 1 and 2, respectively. For class- $i$  jobs, arrivals follow a Poisson process with rate  $\lambda_i$ , while services follow a PH distribution with parameters  $(\alpha^{(i)}, \mathbf{A}^{(i)})$ . We assume a finite buffer for high-priority jobs as its size must be chosen to keep the blocking probability below a certain threshold. Instead, for low-priority jobs we assume the buffer has infinite capacity. We further assume a preemptive scheduling policy, where low-priority jobs start service only when no high-priority jobs are present, and a low-priority job in service is pushed back to the head of its buffer if a high-priority job arrives.

Given the assumptions above, and since only one event occurs at any given time, the number of jobs of either type increases or decreases by one. We can therefore model this queue as a QBD

```

1  /**
2   * Returns the service phase of process
3   * @return Service phase
4   */
5  public int getSrvPhase() {
6      return this.prop[0];
7  }
8
9  /**
10   * @see jmarkov.basic.State#isConsistent()
11   */
12  @Override
13  public boolean isConsistent() {
14      // TODO Complete
15      return true;
16  }
17
18  /**
19   * Returns the service status
20   * @return Service status (1 = busy, 0 = free)
21   */
22  public int getSrvStatus() {
23      return (getSrvPhase() == 0) ? 0 : 1;
24  }
25
26  @Override
27  public HiperExQueueState clone() {
28      return new HiperExQueueState(getSrvPhase());
29  }

```

Figure 8: `rate` method of class `HiperExQueue.java`

where the *level* holds the number of low-priority jobs, while all other information necessary to describe the system state is left for the *phase*. The phase thus holds the number of high-priority jobs in the system and the service phase of the job currently in service. We also include in the phase the type of the job currently in service, which is not strictly necessary but is helpful to describe the model and to extend it. Our first step is therefore to define the system *state* as in the following code snippet.

```

1  class PriorityQueueMPHPHPreemptState extends PropertiesState {
2      public PriorityQueueMPHPHPreemptState(int numberHiJobs, int servicePhase, int serviceType) {
3          super(3);
4          setProperty(0, numberHiJobs);
5          setProperty(1, servicePhase);
6          setProperty(2, serviceType);
7      }
8  }

```

Note that our class `PriorityQueueMPHPHPreemptState` extends the jMarkov abstract class `PropertiesState`, which allows us to define the state as an array of integers. The state is thus defined by three integers that hold the number of high priority jobs, the service phase, and the type of the job in service. Notice that we only need to define the *phase*, as the level behaves as in a QBD, taking values on the non-negative integers and increasing/decreasing by at most one in a single transition. The constructor simply calls the super-class specifying that the phase is described with 3 integers, and sets each of them in their corresponding position.

We now move on to define the *events* via the `PriorityQueueMPHPHPreemptEvent` class as follows.

```

1  class PriorityQueueMPHPHPreemptEvent extends Event {
2      public enum Type {
3          ARRIVAL_HI,
4          SERVICE_END_HI,
5          SERVICE_PHASECHG_HI,
6          ARRIVAL_LOW,
7          SERVICE_END_LOW,
8          SERVICE_PHASECHG_LOW
9      }
10     Type eventType;
11     int eventPhase;
12 }

```

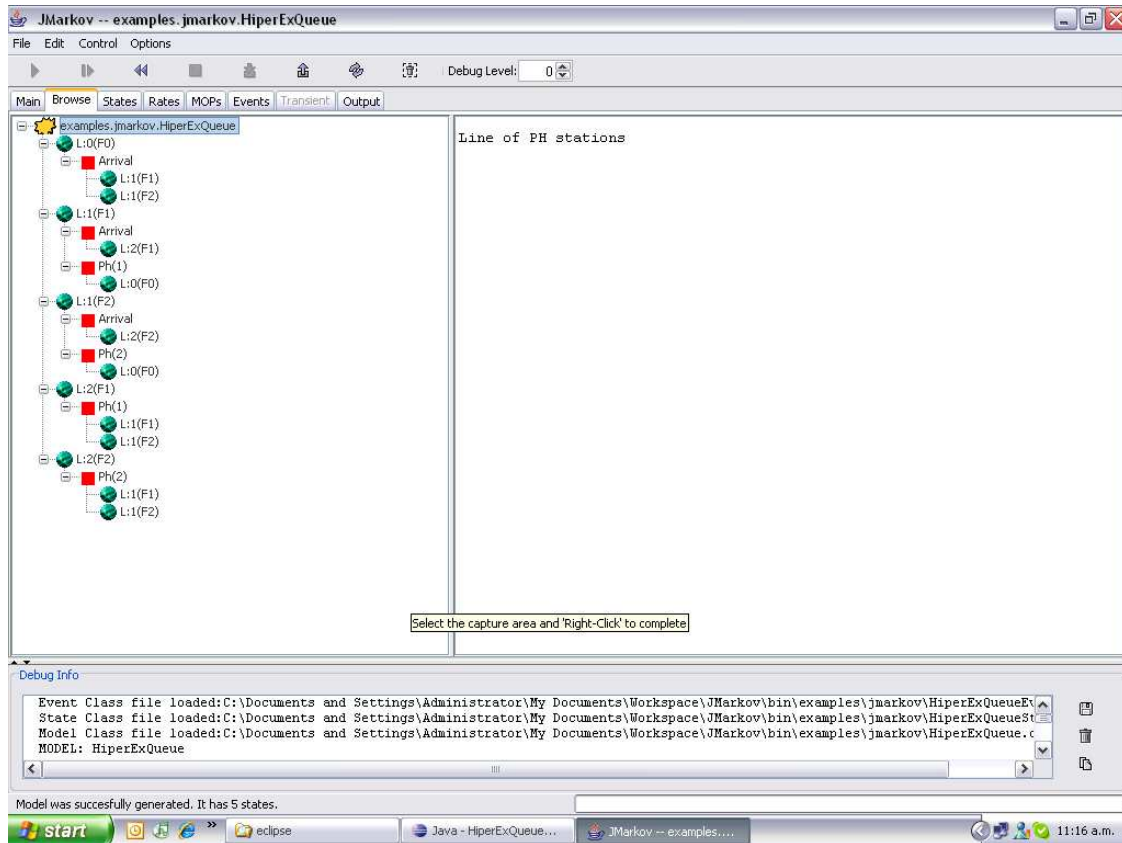


Figure 9: GUI example of jMarkov

1	MEASURES OF PERFORMANCE		
2			
3	NAME	MEAN	SDEV
4			
5	Expected Level	0.14286	?
6	Server Utilization	0.12500	0.33072

Figure 10: MOPs report of jMarkov

Here we see that this class extends the abstract class `Event` and defines an enumeration `Type` to list all the possible events: arrivals, service completion, and service phase change without completion, for both high and low priority jobs. Lines 10-11 then show that the two properties that define an event are the type of the event, and the *service phase* in which the event occurs. Note that here by phase we refer to the phase of the job in service, which we set to 0 if the system is idle.

With the definition of states and events we then define our main class `PriorityQueueMPPHPPreempt`, which, as shown in the following snippet, extends the `GeomProcess` class since our model is a QBD.

```

1 public class PriorityQueueMPPHPPreempt extends
2   GeomProcess<PriorityQueueMPPHPPreemptState, PriorityQueueMPPHPPreemptEvent>{
3   double lambda_hi;
4   double lambda_low;
5   PhaseVar servTime_hi;
6   PhaseVar servTime_low;
7   int bufferCapacity;
8 }

```

Here lines 2 and 3 define the properties associated to the arrival rates, while lines 4 and 5 define the PH variables that describe the service process. These are `jPhase` objects. The final property is the capacity of the high-priority buffer. As part of this class we need to define the `active`, `dests`, and `rates` methods. The following code illustrates part of the `active` method.

```

1 switch (event.eventType) {
2     case ARRIVAL_HI:
3         if ( state.getNumberHiJobs() < bufferCapacity )
4             result = true;
5         break;
6     case SERVICE_END_HI:
7         result = (state.getServiceType()==1 && state.getServicePhase() == event.eventPhase);
8         result = result && servTime_hi.getMat0().get(state.getServicePhase()-1) > 0;
9         break;
10 }

```

In case the event is a high-priority arrival, lines 3-5 allow it to be active if there is spare capacity in the buffer. Instead, if the event is a high-priority service completion, line 7 first checks if the current job in service is of class 1 and if its service phase matches that of the event. Next, line 8 checks if it is actually possible to have a service completion in such phase, i.e., if the entry of the exit vector  $-\mathbf{A}^{(1)}\mathbf{1}$  corresponding to the current service phase is positive. This vector is obtained with the `jPhase` `getMat0` method. Similar checks are performed for all other events.

Next, in the `dests` and `rate` methods we define the destination state for each event in each state, and the corresponding transition rate. In the interest of space, the next snippet depicts a small section of the `rate` method, where we define the transition rate in case of a high-priority arrival.

```

1 switch (event.eventType) {
2     case ARRIVAL_HI:
3         if (curState.getNumberHiJobs() == 0){
4             rate = lambda_hi*servTime_hi.getVector().get(newPhase-1);
5         }else
6             rate = lambda_hi;
7         break;
8     ...
9 }

```

Here lines 3-4 consider the case where the number of high-priority jobs in the current state is zero, which allows the new high-priority job to start service, even if a low-priority job is present. The transition rate is then the arrival rate times the probability that a new high-priority service starts in the phase marked by the destination state. This probability is obtained with the `jPhase` `getVector` method. Instead, lines 5-6 cover the case where a high-priority job is already in service, thus the new job simply joins the queue with transition rate given by its arrival rate.

With all the previous definitions we now state the `main` method, where we set up the parameters of the model, and call the `jMarkov` routines to build the model, solve it, and compute the measures of performance, as shown next.

```

1 public static void main(String[] a) {
2     double lambda_hi = 0.2;
3     double lambda_low = 0.2;
4
5     double[] data = readTextFile("src/examples/jphase/W2.txt");
6     EMHyperErlangFit fitter_hi = new EMHyperErlangFit(data);
7     ContPhaseVar servTime_hi = fitter_hi.fit(4);
8
9     MomentsACPHFit fitter_low = new MomentsACPHFit(2, 6, 25);
10    ContPhaseVar servTime_low = fitter_low.fit();
11
12    int bufferCapacity = 100;
13    PriorityQueueMPPHPPreempt model = new PriorityQueueMPPHPPreempt(lambda_hi, lambda_low,
14        servTime_hi, servTime_low, bufferCapacity);
15    model.generate();
16    model.printMOPs();
17 }

```

Here lines 2-3 define the arrival rates of both job types. Next, lines 5-7 build the PH distribution for the high-priority services. To this end, we first read a data trace into a double array, which we pass to a `jPhase` `EMHyperErlangFit` fitter to obtain the fitted PH distribution. Lines 9-10 perform a similar step, but in this case we use a moment-matching method to obtain a low-priority service-time PH distribution with a given set of first three moments. After this, line 12 defines the buffer capacity and line 13 builds the model object with all the parameters. Lines 15-16 ask `jMarkov` to generate the model and compute the measures of performance, and we obtain the following result.

1	MEASURES OF PERFORMANCE		
2	NAME	MEAN	SDEV
3	Expected Level	6.47779	
4	Number High Jobs	1.02821	1.79758
5	High Jobs Blocking Probability	0.00494	0.07010
6	Utilization	0.84043	0.36621

Thus, with the parameters as above, the mean number of high and low priority jobs is 1.02 and 6.47, respectively, while the blocking probability of high-priority jobs is 0.0049. The output also includes the mean server utilization and the standard deviation of the performance measures.

We highlight three central takeaways from the above example. (i) The definition of the model is made at a high level, referring to events (arrivals, service completions, service phase transitions), and their effect on the system state. At no point one needs to explicitly define the entries of the matrices  $\mathbf{A}_0$ ,  $\mathbf{A}_1$ , or  $\mathbf{A}_2$  in (5.1), which is not a trivial task when the model is made of several variables as in this example. jMarkov takes care of this task. (ii) Once the model is defined, it is relatively simple to introduce a modification in the operational rules. Consider for instance modifying the preemptive policy by a non-preemptive one. If one is in charge of building the transition matrix (5.1), this would require an almost completely new model. Instead, with jMarkov we can start with the current model and modify the `dests` and `rate` methods, specifically the cases where a high priority arrival occurs. This facilitates the evaluation of different policies, which is a common task in system modeling. (iii) The integration of the jQBD and jPhase modules allows us to use the representation of PH variables when defining the QBD model with the `active`, `dests`, and `rate` methods. In these methods we can explicitly refer to the initial phase probabilities, or to the rates of service completion at any given phase. Further, we can exploit the fitting methods in jPhase to define the model parameters, using either trace data or statistics such as the mean or variance. The integration of these modules in jMarkov thus facilitates the development and evaluation of complex models.

## 7 Further Development

This project is currently under development, and therefore we appreciate all the feedback we can receive.

## References

- [1] G. Ciardo. Tools for formulating Markov models. In W. K. Grassman, editor, *Computational Probability*. Kluwer's International Series in Operations Research and Management Science, Massachusetts, USA, 2000.
- [2] B. Heimsund. Matrix Toolkits for Java (MTJ), December 2005. Last modified: Monday, 05-Dec-2005 09:03:23 CET.
- [3] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. JAMA: A java matrix package, July 2005. MathWorks and the National Institute of Standards and Technology (NIST).
- [4] jMarkov website. Available online at <https://projects.coin-or.org/jMarkov/>, 2016.
- [5] V. Kulkarni. *Modeling and analysis of stochastic systems*. Chapman & Hall., 1995.
- [6] G. Latouche and V. Ramaswami. *Introduction to matrix analytic methods in stochastic modeling*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1999.



- [7] M. F. Neuts. *Matrix-geometric solutions in stochastic models*. The John Hopkins University Press, 1981.
- [8] J. F. Pérez and G. Riaño. jPhase: an object-oriented tool for modeling Phase-Type distributions. In *SMCtools '06: Proceedings from the 2006 Workshop on Tools for Solving Structured Markov Chains*, New York, 2006. ACM Press.
- [9] J. F. Pérez and G. Riaño. *jPhase User's Guide*. Universidad de los Andes, 2006.
- [10] G. Riaño and J. Góez. *jMarkov User's Guide*. Industrial Engineering, Universidad de los Andes, 2005.
- [11] G. Riaño and A. Sarmiento. jMDP: an object-oriented framework for modeling MDPs. Working paper. Universidad de los Andes, 2006.
- [12] A. Sarmiento and G. Riaño. *jMDP User's Guide*. Industrial Engineering, Universidad de los Andes, 2005.
- [13] Sun Microsystems. Java technology, Jan. 2006.
- [14] P. van der Linden. *Just Java(TM) 2*. Prentice Hall, 6th edition, 2004.