jMarkov User's Guide

Germán Riaño, Julio Góez, and Juan F. Pérez

Contents

1 Introduction 2

2 Building Large - Scale Markov Chains 2
2.1 Space state building algorithm . . . 2
2.2 Measures of Performance . . . 3

3 Framework Design 4
3.1 Java and Object Oriented Programming . . . 4
3.2 Build Package . . . 4
3.3 Basic Package . . . 4
3.4 The Solvers Package . . . 6

4 Examples 6
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

5 Modeling Quasi-Birth and Death Processes 25
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

6 Modeling Priority Queues: incorporating phase-type distributions with jPhase 28

7 Further Development 32

References 32

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

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

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

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

This matrix can be computed as

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

For an irreducible chain, the stationary distribution $\boldsymbol{\pi} = [\pi_1, \pi_2, \ldots,]$ 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
      $\mathcal{D} :=$ 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
        $R_{ij} := R_{ij} +$ 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

$$E\big(r(X(t)\big) = \mathbf{a}\mathbf{P}(t)\mathbf{r},$$

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

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

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.

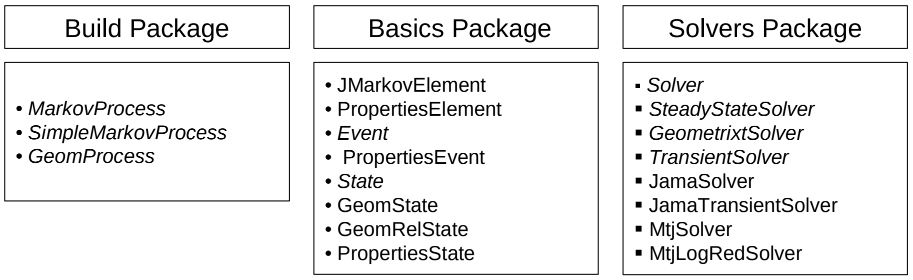


Figure 2: Class classification

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 `dests`. 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`, `dests`, 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,

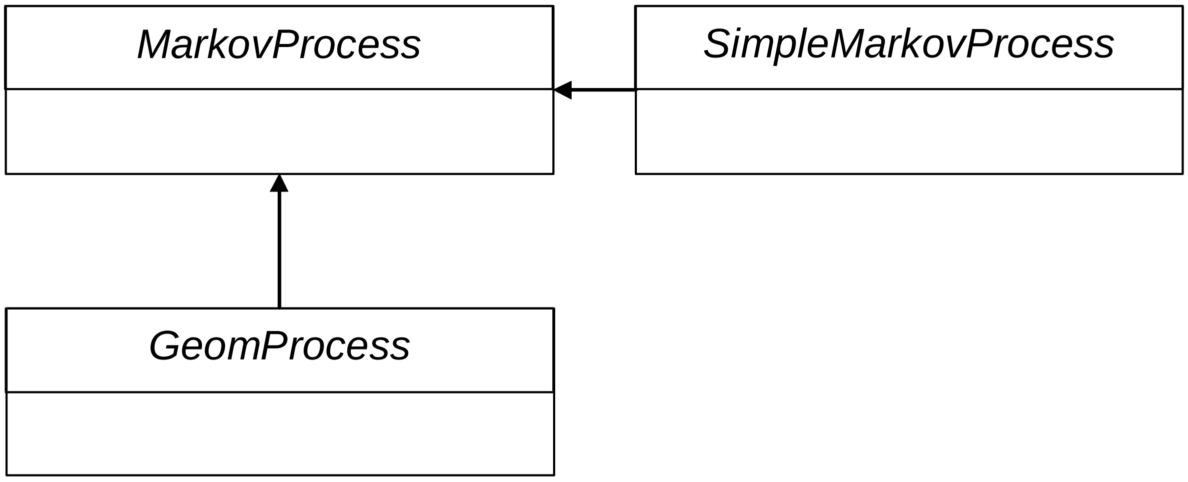


Figure 3: Class diagram build module

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.

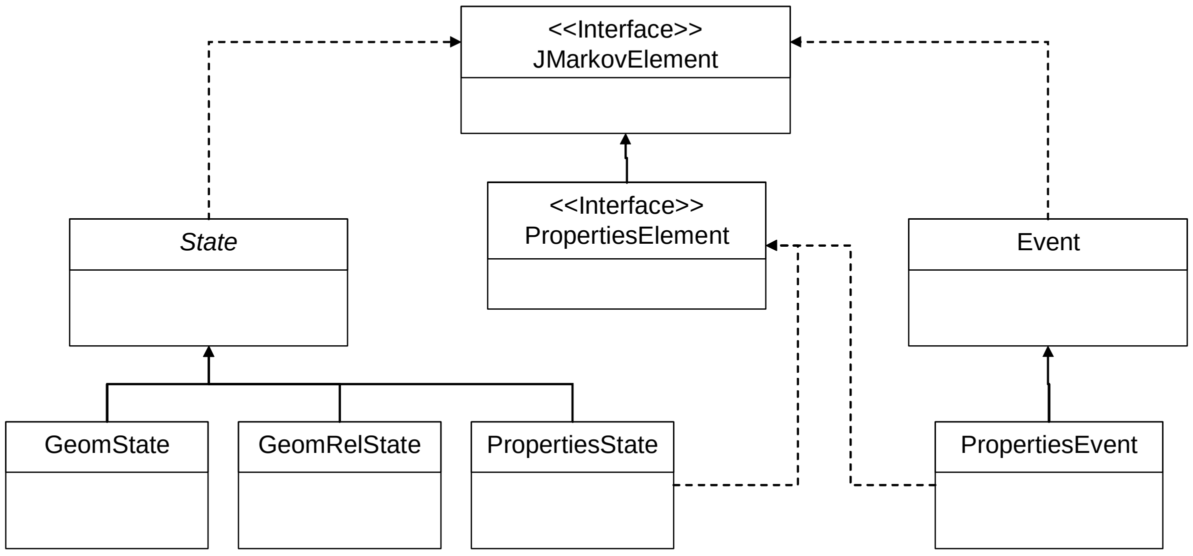


Figure 4: Class diagram for the basic package

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 relay on two popular Java packages to handle matrix operations JAMA [3] and MTJ [2], for dense and sparse matrices, respectively.

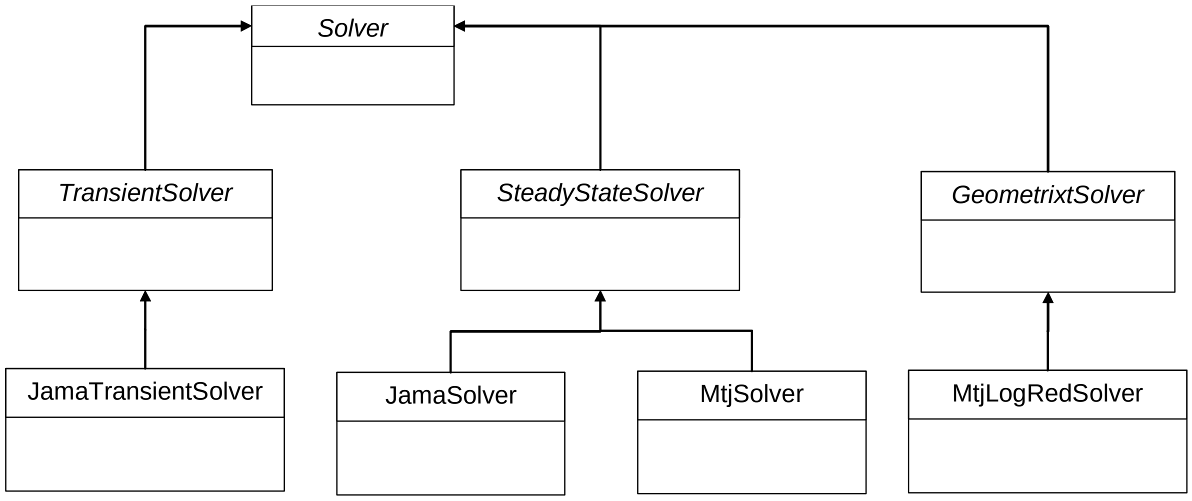


Figure 5: Class diagram of the solvers package

4 Examples

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

Assume that a system has Poisson arrivals with rate λ. There are two exponential servers with rates μ_1 and μ_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 α. Otherwise he will pick he 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

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

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	μ_2			λ					
100	μ_1			λ					
110		μ_1	μ_2		λ				
111						λ			
112					$\mu_1+\mu_2$				
$\vdots$									
1,1,N-3									λ
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

```java
package examples.jmarkov;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import jmarkov.MarkovProcess;
import jmarkov.SimpleMarkovProcess;
import jmarkov.basic.Event;
import jmarkov.basic.EventsSet;
import jmarkov.basic.PropertiesState;
import jmarkov.basic.States;
import jmarkov.basic.StatesSet;

/**
 * This class represents a system with 2 different exponential
 * servers with rates mu1 and mu2, respectively, and arrival rate
 * lambda.
 * @author Germán Riaño. Universidad de los Andes.
 */

public class QueueMM2dN extends SimpleMarkovProcess<MM2dNState, QMM2dNEvent> {
    // Events
    final int ARRIVAL = 0;
    final int ARRIVAL1 = 1; // only for empty system
    final int ARRIVAL2 = 2; // only for empty system
    final int DEPARTURE1 = 3;
    final int DEPARTURE2 = 4;
    private double lambda;
    private double mu1, mu2, alpha;
    private int N;

    /**
     * Constructs a M/M/2d queue with arrival rate lambda and service
     * rates mu1 and mu 2.
     * @param lambda Arrival rate
     * @param mu1 Server 1 rate
     * @param mu2 Server 2 rate
     * @param alpha Probability of an arriving customer choosing
     *        server 1 (if both idle)
     * @param N Max number in the system
     */
    public QueueMM2dN(double lambda, double mu1, double mu2, double alpha, int N) {
        super((new MM2dNState(0, 0, 0)), //
                QMM2dNEvent.getAllEvents()   ); // num Events
        this.lambda = lambda;
        this.mu1 = mu1;
        this.mu2 = mu2;
        this.alpha = alpha;
        this.N = N;
    }

    /**
     * Returns an QueueMM2N object with arrival rate 4.0, service rate
     * of the first server 2.0, service rate of the second server 3.0,
     * probability of choose the first server 0.3 and capacity of 8
     * customers in the system. Used by GUI
     */
    public QueueMM2dN() {
        this(1.0, 2.0, 3.0, 0.3, 8);
    }

    /**
     * Determines the active events
     */
    public @Override boolean active(MM2dNState i, QMM2dNEvent e) {
        boolean result = false;
        switch (e.getType()) {
        case ARRIVAL:
            result = ((i.getQSize() < N - 2) && (!i.isEmpty()));
            break;
        case ARRIVAL1:
            result = i.isEmpty();
            break;
        case ARRIVAL2:
            result = i.isEmpty();
            break;

        case DEPARTURE1:
            result = (i.getStatus1() > 0);
            break;
        case DEPARTURE2:
            result = (i.getStatus2() > 0);
            break;
        }
        return result;
    }

    public @Override States<MM2dNState> dests(MM2dNState i, QMM2dNEvent e) {
        int newx = i.getStatus1();
        int newy = i.getStatus2();
        int newz = i.getQSize();

        switch (e.getType()) {
        case ARRIVAL:
            if (i.getStatus1() == 0) {
                newx = 1;
            } // serv 1 desocupado
            else if (i.getStatus2() == 0) {
                newy = 1;
            } // serv 2 desocupado
            else { // ambos ocupados
                newz = i.getQSize() + 1;
            }
            break;
        case ARRIVAL1:
            newx = 1;
            break;
        case ARRIVAL2:
            newy = 1;
            break;
        case DEPARTURE1:
            if (i.getQSize() != 0) {
                newx = 1;
                newz = i.getQSize() - 1;
            } else {
                newx = 0;
            }
            break;
        case DEPARTURE2:
            if (i.getQSize() != 0) {
                newy = 1;
                newz = i.getQSize() - 1;
            } else {
                newy = 0;
            }
            break;
        }
        return new StatesSet<MM2dNState>( new MM2dNState(newx, newy, newz));
    }

    public @Override double rate(MM2dNState i,MM2dNState j, QMM2dNEvent e) {
        double res = 0;
        switch (e.getType()) {
        case ARRIVAL:
            res = lambda;
            break;
        case ARRIVAL1:
            res = lambda * alpha;
            break;
        case ARRIVAL2:
            res = lambda * (1 - alpha);
            break;
        case DEPARTURE1:
            res = mu1;
            break;
        case DEPARTURE2:
            res = mu2;
            break;
        }
        return res;
    }

    @Override
    public String description() {
        return "M/M/2/N SYSTEM\nQueueing System with two servers, with rates "
                + mu1 + " and " + mu2 + ".\nArrivals are Poisson with rate "
                + lambda + ",\nand the maximum number in the system is " + N;

    }

    /**
     * This method just tests the class.
     * @param a Not used
     */
    public static void main(String[] a) {
        String stg;
        BufferedReader rdr = new BufferedReader(
                new InputStreamReader(System.in));
        try {
            System.out.println("Input rate ");
            stg = rdr.readLine();
            double lda = Double.parseDouble(stg);
            System.out.println("Service rate 1  ");
            stg = rdr.readLine();
            double mu1 = Double.parseDouble(stg);
            System.out.println("Service rate 2  ");
            stg = rdr.readLine();
            double mu2 = Double.parseDouble(stg);
            System.out.println("Provide alpha  ");
            stg = rdr.readLine();
            double alpha = Double.parseDouble(stg);
            System.out.println("Max in the system ");
            stg = rdr.readLine();
            int N = Integer.parseInt(stg);
            QueueMM2dN theQueue = new QueueMM2dN(lda, mu1, mu2, alpha, N);
            theQueue.showGUI();
            theQueue.printAll();
        } catch (IOException e) {
        }
        ;
    }

} // class end

/**
 * This is a particular case of propertiesState, whith three
 * properties, namely the server 1 and 2 status, plus the queue level.
 * @author Germán Riaño. Universidad de los Andes.
 */

class MM2dNState extends PropertiesState {
```

```java

    /**
     * We identify each State with the triplet (x,y,z), where x and y
     * are the status of the servers and z the number in queue (0,1,
     * ..,N-2).
     */

    MM2dNState(int x, int y, int z) {
        super(3); // Creates a PropertiesState with 3 properties.
        this.prop[0] = x;
        this.prop[1] = y;
        this.prop[2] = z;
    }

    @Override
    public void computeMOPs(MarkovProcess mp) {
        setMOP(mp, "Status_Server_1", getStatus1());
        setMOP(mp, "Status_Server_2", getStatus2());
        setMOP(mp, "Queue_Length", getQSize());
        setMOP(mp, "Number_in_System", getStatus1() + getStatus2() + getQSize());
    }

    /**
     * Returns the status of the first Server
     * @return Status of the first Server
     */
    public int getStatus1() {
        return prop[0];
    }

    /**
     * Returns the status of the second Server
     * @return Status of the second Server
     */
    public int getStatus2() {
        return prop[1];
    }

    /**
     * Returns the size of the queue
     * @return Status of the size of the queue
     */
    public int getQSize() {
        return prop[2];
    }

    /*
     * isEmpty detects is the system is empty. It comes handy when
     * checking whether the events ARRIVAL1 and ARRIVAL2 are active.
     */
    boolean isEmpty() {
        return (getStatus1() + getStatus2() + getQSize() == 0);
    }

    /**
     * @see jmarkov.basic.State#isConsistent()
     */
    @Override
    public boolean isConsistent() {
        // TODO Complete
        return true;
    }

    /*
     * We implement label so that States are labeld 1, 1A, 1B, 2, 3,
     * ..., N-2
     */
    @Override
    public String label() {
        String stg = "0";
        if ((getStatus1() == 1) && (getStatus2() == 0))
            stg = "1A";
        if ((getStatus2() == 1) && (getStatus1() == 0))
            stg = "1B";
        if ((getStatus2() == 1) && (getStatus1() == 1))
            stg = "" + (2 + getQSize());
        return stg;
    }

    /*
     * This method gives a verbal description of the State.
     */
    @Override
    public String description() {
        String stg = "";
        stg += "Server 1 is " + ((getStatus1() == 1) ? "busy" : "idle");
        stg += ". Server 2 is " + ((getStatus2() == 1) ? "busy" : "idle");
        stg += ". There are " + getQSize() + " customers waiting in queue.";
        return stg;
    }

}

class QMM2dNEvent extends Event {
    /** Event types */
    public enum Type {
        /** An arrival */
        ARRIVAL,
        /** Arrival to server 1 (only for emtpy system) */
        ARRIVAL1,
        /** Arrival to server 2 (only for emtpy system) */
        ARRIVAL2,
        /** departure from server 1 */
        DEPARTURE1,
        /** departure from server 2 */
        DEPARTURE2;
    }

    private Type type;

    /**
     * @param type
     */
    public QMM2dNEvent(Type type) {
        super();
        this.type = type;
    }

    /**
     * @return Returns the type.
     */
    public final Type getType() {
        return type;
    }

    /**
     * @return the set of all events.
     */
    public static EventsSet<QMM2dNEvent> getAllEvents() {
        EventsSet<QMM2dNEvent> evSet = new EventsSet<QMM2dNEvent>();
        for (Type type : Type.values())
            evSet.add(new QMM2dNEvent(type));
        return evSet;
    }
}

// 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 $\mu_1, \mu_2, \ldots, \mu_K$. 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 α_k, i.e., he will choose server k with probability

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

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 $S(t) = (S_1(t), \ldots, S_K(t))$, 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

```java
package examples.jmarkov;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import jmarkov.MarkovProcess;
import jmarkov.SimpleMarkovProcess;
import jmarkov.basic.Event;
import jmarkov.basic.EventsSet;
import jmarkov.basic.PropertiesState;
import jmarkov.basic.States;
import jmarkov.basic.StatesSet;

/**
 * This class represents is a system with K different
 * exponential servers with rates mu1, mu2, etc,
 * respectively, and arrival rate lambda. A customer
 * that finds more then one server idle chooses according
 * to relative intensities
 * <tex txt="$\alpha_1, \alpha_2, \ldots, \alpha_K$">
 * alpha1, alpha2, etc</tex>. The probability of choosing
 * idle server k will be given by
 * <tex txt="\[\beta_k = \frac{\alpha_k}{\sum_{\ell\in \cal I} \alpha_{\ell}},\]
 * where $\cal I$ is the set of idle servers.">
 * alpha(k) / sum( alpha(j)), where the sum is over the set of idle servers.
 * </tex>
 * @author Germán Riaño. Universidad de los Andes.
 */
public class QueueMMKdN extends SimpleMarkovProcess<QueueMMKdNState,QueueMMKdNEvent> {
        // Events

        private double lambda;
        private double[] mu, alpha;
        private int K; // number of servers
        private int N;
        private static final int NDARRIVAL = QueueMMKdNEvent.NDARRIVAL;
        private static final int DIRARRIVAL = QueueMMKdNEvent.DIRARRIVAL;
        private static final int DEPARTURE = QueueMMKdNEvent.DEPARTURE;

        /**
         * Constructs a M/M/Kd queue with arrival rate lambda and service
         * rates mu, relative probabilities of choosing each server alpha
         * @param lambda  Arrival rate
         * @param mu      Server  rates
         * @param alpha   Relative probability of an arriving customer choosing each server.
         * @param N       Max number in the system
         */
        public QueueMMKdN(double lambda, double[] mu, double[] alpha, int N) {
                super(
                        new QueueMMKdNState(mu.length, alpha),
                        QueueMMKdNEvent.getAllEvents(mu.length));
                this.K = mu.length;
                this.lambda = lambda;
                this.mu = mu;
                this.alpha = alpha;
                this.N = N;
        }

    /**
     * Returns an QueueMMKdN object with arrival rate 1.0,
     * service rates of 2.0, 3.0 and 4.0;
     * and capacity of 8 customers in the system.
     * Used by GUI
     */
        public QueueMMKdN(){
                this(1.0, new double[]{2,3,4}, new double[]{2,3,4}, 8);
        }

        /**
         * Determines the active events.
         */
        @Override
        public boolean active(QueueMMKdNState i, QueueMMKdNEvent e) {
                boolean result = false;
                switch (e.type) {
                        case (NDARRIVAL) : // NDARIIVAL occurs only if servers are busy and there is roon in the
                                result = (i.allBusy() && (i.getQSize() < N - K));
                                break;
                        case (DIRARRIVAL) :
                                {
                                        result = (i.getStatus(e.server) == 0);
                                        //DirARRIVAL occurs if server is EMPTY.
                                        break;
                                }
                        case (DEPARTURE) :
                                { // ev.type == DEPARTURE
                                        result = (i.getStatus(e.server) == 1);
                                        //DEPARTURE occurs if server is busy.
                                }
                }
                return result;
        }

        /*
         * Determines the possible destination event (actually one in this case).
         */

        @Override
        States<QueueMMKdNState> dests(QueueMMKdNState i, QueueMMKdNEvent e) {
                int[] status = new int[K];
                for (int k = 0; k < K; k++)
                        status[k] = i.getStatus(k); //copy current values
                int Q = i.getQSize();
                switch (e.type) {
                        case (NDARRIVAL) :
                                Q++; // non-directed ARRIVAL
                                break;
                        case (DIRARRIVAL) :
                                status[e.server] = 1; //directed ARRIVAL, picks a server.
                                break;
                        case (DEPARTURE) :
                                if (Q > 0) { //there is Queue
                                        status[e.server] = 1; //set (keeps) server busy
                                        Q--; // reduce queue
                                } else
                                        status[e.server] = 0; //set server idle
                }
                return new StatesSet<QueueMMKdNState>(new QueueMMKdNState(status, Q, alpha));
        }

        /*
         * The rate is lambda, or mu for non-directed arrival and for departure.
         * For directed arrival rate id lambda 8 prob(server is choosen)
         * @see jmarkov.SimpleMarkovProcess#rate(jmarkov.State, jmarkov.State, jmarkov.Event)
         */
        @Override
        public double rate(QueueMMKdNState i, QueueMMKdNState j, QueueMMKdNEvent e) {
                double result = 0;

                switch (e.type) {
                        case (DEPARTURE) :
                                result = mu[e.server];
                                break;
                        case (NDARRIVAL) :
                                result = lambda;
                                break; //non-directed arrival
                        case (DIRARRIVAL) :
                                result = i.prob(e.server) * lambda;
                }
                return result;
        }

        /**
         * Main Method. This asks the user for parameters
     * and tests the program.
     * @param a Not used
         */
        public static void main(String[] a) {
                BufferedReader rdr =
                        new BufferedReader(new InputStreamReader(System.in));
                try {
                        System.out.println("Input Rate: ");
                        double lda = Double.parseDouble(rdr.readLine());
                        System.out.println("Num Servers: ");
                        int K = Integer.parseInt(rdr.readLine());
                        double mu[] = new double[K];
                        double alpha[] = new double[K];
                        for (int k = 0; k < K; k++) {
                                System.out.println("Service rate, server " + (k + 1) + " : ");
                                mu[k] = Double.parseDouble(rdr.readLine());
                        }
                        for (int k = 0; k < K; k++) {
                                System.out.println(
                                                "Choosing intensity, server  " + (k + 1) + " : ");
                                alpha[k] = Double.parseDouble(rdr.readLine());
                        }
                        System.out.println("Max in system : ");
                        int N = Integer.parseInt(rdr.readLine());
                        QueueMMKdN theModel = new QueueMMKdN(lda, mu, alpha, N);
                        theModel.showGUI();
                        //theModel.setDebugLevel(2);
                        theModel.printAll();
                } catch (IOException e) {
                };
        }

        /**
         * @see jmarkov.SimpleMarkovProcess#description()
         */
        @Override
        public String description() {
                String stg = "M/M/k/N SYSTEM\n\n";
                stg += "Multiple server queue with " + this.K + " different servers\n";
                stg += "Arrival Rate = " + lambda + ", Max number in system " + N;
                return stg;
        }

} //class end
/**
 * This is a particular case of propertiesState, whith K + 1
 * properties, namely the server 1, 2, ..., K status, plus the queue level.
 *
 * @author Germán Riaño. Universidad de los Andes.
 */
class QueueMMKdNState extends PropertiesState {

        private int K; // number of servers
        private double sumProb = -1; // sum of relative probabilities
        private double[] alpha; //relative frequency of servers
        private double[] beta; //probabilities for this state
        /**
         * Constructs a state for an empty system with K servers, and
         * choosing intensities alpha.
         * @param K Number of servers.
         */
        QueueMMKdNState(int K, double[] alpha) {
                this(new int[K], 0, alpha);
        }

        /**
         * We identify each State with a vector that counts the
         * ststus fo the k servers and
         * the number in queue (0,1, ..,N-K).
         */
        QueueMMKdNState(int[] status, int Qsize, double[] alpha) {
                super(alpha.length + 1);
                this.K = alpha.length;
                this.alpha = alpha;
                this.beta = new double[K];
                int sum = 0; // adds the number of busy server = people in service
                for (int i = 0; i < K; i++) {
                        prop[i] = status[i];
                        sum += status[i];
                }
                prop[K] = Qsize;
        }

        /**
         * Computes the MOPs
         * @see jmarkov.basic.State#computeMOPs(MarkovProcess)
         */
        @Override
        public void computeMOPs(MarkovProcess mp) {
                double sum = 0.0;
                for (int i = 0; i < K; i++) {
                        sum += getStatus(i);
                        setMOP(mp,"Server Status " + (i + 1), getStatus(i));
                }
                setMOP(mp,"Queue Length", getQSize());
                setMOP(mp,"Number in System", sum + getQSize());
        }
```

```

    /**
     * Returns the status of the kth Server
     * @param k server index
     * @return Status of the kth Server
     */
        public int getStatus(int k) {
                return prop[k];
        }

    /**
     * Returns the size of the queue
     * @return Status of the size of the queue
     */
        public int getQSize() {
                return prop[K];
        }
        /**
         * Determines if all servers are busy
     * @return True, if all servers are busy. False, otherwise
         */
        public boolean allBusy() {
                boolean result = true;
                for (int k = 0; result && (k < K); k++)
                        result = result && (getStatus(k) == 1);
                return result;
        }
    /**
     * Determines if all servers are idle
     * @return True, if all servers are idle. False, otherwise
     */
        public boolean allIdle() {
                boolean result = true;
                for (int k = 0; result && (k < K); k++)
                        result = result && (getStatus(k) == 0);
                return result;
        }
    /**
     * @see jmarkov.basic.State#isConsistent()
     */
    @Override
    public boolean isConsistent() {
        // TODO Complete
        return true;
    }
        /*
         * determines the sum of all intensities for idle servers. The result
         * is kept in sumProb for future use.
         */
        private double sum() {
                if (sumProb != -1)
                        return sumProb;
                double res = 0;
                for (int k = 0; k < K; k++) {
                        res += (1 - getStatus(k)) * alpha[k];
                }
                return (sumProb = res);
        }
        /**
         * Detemines the probability of an idle server being choosen
         * among idle servers. A customer that finds more then one server
         *  idle chooses according to relative intensities
         * <tex txt="$\alpha_1, \alpha_2, \ldots, \alpha_K$">
         * alpha1, alpha2, etc</tex>. The probability of choosing idle
         *  server k will be given by
         * <tex txt="\[\beta_k = \frac{\alpha_k}{\sum_{\ell\in \cal I} \alpha_{\ell}},\]
         * where $\cal I$ is the set of idle servers.">
         * alpha(k) / sum(j, alpha(j)), where the sum is over the set
         * of idle servers. </tex>
     * @param server server index
     * @return probability of an idle server being choosen
     * among idle servers
         */
        public double prob(int server) {
                if (beta[server] != 0)
                        return beta[server];
                return (
                        beta[server] = (((1 - getStatus(server)) * alpha[server]) / sum()));
        }

        /**
         * Returns a label with the format SxxQz, whre xx is the list of busy servers.
         * @see jmarkov.basic.State#label()
         */
        @Override
```

```java
	public String label() {
		String stg = "S";
		for (int k = 0; k < K; k++) {
			stg += (getStatus(k) == 1) ? "" + (k + 1) : "";
		}
		return stg + "Q" + getQSize();
	}

	/*
	 * This method gives a verbal description of the State.
	 */
	@Override
	public String description() {
		String stg = "";
		if (!allIdle())
			stg += "Busy Servers:";
		else
			stg += "No one in service";
		for (int k = 0; k < K; k++) {
			stg += (getStatus(k) == 1) ? "" + (k + 1) + "," : "";
		}
		stg += " There are " + getQSize() + " customers waiting in queue.";
		return stg;
	}

}
/**
	 *
	 * This class define the events.
	 * An event has two components: type which can have three values
	 * depending whether it represents a directed arrival, a
	 * non-directed arrival or a departure, and server, which
	 * represents the choosen server (if arrival) or the finishing
	 * server. For non-directed arrivals we set server -1 by convention.
	 *
	 * @author Germán Riaño
	 *
	 */
class QueueMMKdNEvent extends Event {
	final static int NDARRIVAL = 0;
	//Non directed arrival (when all servers are busy)
	final static int DIRARRIVAL = 1; //Directed arrival chooses among server(s)
	final static int DEPARTURE = 2;
	int type; // ARRIVAL or DEPARTURE
	/*	server = chosen server if ARRIVAL finds many available,
	 * server = -1 if no server available
	 * server = finishing server if DEPARTURE event
	 */
	int server;
	QueueMMKdNEvent(int type, int server) {
		this.type = type;
		this.server = server;
	}

	static EventsSet<QueueMMKdNEvent> getAllEvents(int K) {
		EventsSet<QueueMMKdNEvent> eSet = new EventsSet<QueueMMKdNEvent>();
		eSet.add(new QueueMMKdNEvent(NDARRIVAL, -1));
		for (int i = 0; i < K; i++) {
			eSet.add(new QueueMMKdNEvent(DIRARRIVAL, i));
		}
		for (int i = 0; i < K; i++) {
			eSet.add(new QueueMMKdNEvent(DEPARTURE, i));
		}
		return eSet;
	}

	/* (non-Javadoc)
	 * @see java.lang.Object#toString()
	 */
	@Override
	public String label() {
		String stg = "";
		switch (type) {
			case (NDARRIVAL) :
				stg += "Non-directed arrival";
				break;
			case (DIRARRIVAL) :
				stg += "Directed arrival to server " + (server + 1);
				break;
			case (DEPARTURE) :
				stg += "Departure from server " + (server + 1);
				break;
		}
		return stg;
	}
```

```

} //end class
// Lets start defining the State

// Now we define the main  class
```

4.3 Drive Thru

4.3.1 Code

```
package examples.jmarkov;

import static examples.jmarkov.DriveThruEvent.Type.ARRIVAL;
import static examples.jmarkov.DriveThruEvent.Type.MIC_COMPLETION;
import static examples.jmarkov.DriveThruEvent.Type.SERVICE_COMPLETION;
import static examples.jmarkov.DriveThruState.CustStatus.BLOCKED_DONE;
import static examples.jmarkov.DriveThruState.CustStatus.COOKING;
import static examples.jmarkov.DriveThruState.CustStatus.EMPTY;
import static examples.jmarkov.DriveThruState.CustStatus.ORDERING;
import static examples.jmarkov.DriveThruState.CustStatus.WAIT_MIC;

import java.io.PrintWriter;

import jmarkov.MarkovProcess;
import jmarkov.SimpleMarkovProcess;
import jmarkov.basic.Event;
import jmarkov.basic.EventsSet;
import jmarkov.basic.State;
import jmarkov.basic.States;
import jmarkov.basic.StatesSet;
import jmarkov.basic.exceptions.NotUnichainException;
import examples.jmarkov.DriveThruState.CustStatus;

/**
 * This class implements a Drive Thru. Extends
 * SimpleMarkovProcess.
 *
 * @author Margarita Arana y Gloria Díaz.  Universidad de los Andes.
 * Mod: Germán Riaño (2004)
 * @version 1.0a
 */
public class DriveThru extends
        SimpleMarkovProcess<DriveThruState, DriveThruEvent> {

    double lambda; // arrival rate
    double mu1; // Service rate for server 1
    double mu2; // Service rate for server 2
    int M; // Maximum number of clients in the system
    int S; // Number of servers
    int N; // Number of places between the window and the microphone

    /**
     * Constructor de un DriveThru.
     *
     * @param lambda
     *            Tasa de arribos
     * @param mu1
     *            Tasa de servicios del micrï¿½fono
     * @param mu2
     *            Tasa de servicios de la ventana
     * @param M
     *            Nï¿½mero mï¿½ximo de entidades en el sistema
     * @param S
     *            Nï¿½mero de servidores
     * @param N
     *            Nï¿½mero de puestos entre la ventana y el micrï¿½fono
     */
    public DriveThru(double lambda, double mu1, double mu2, int M, int S, int N) {
        super((new DriveThruState(N, S)), DriveThruEvent.getAllEvents(N));
        this.lambda = lambda;
        this.mu1 = mu1;
        this.mu2 = mu2;
        this.M = M;
        this.S = S;
        this.N = N;

    }

    /**
     * Default constructor for GUI.
     */
    public DriveThru() {
        this(80.0, 12.0, 30.0, 4, 2, 1);
    }

    /**
     * Determines when the states are active for each state.
     *
     * @see SimpleMarkovProcess#active(State, Event)
     */

    @Override
    public boolean active(DriveThruState s, DriveThruEvent ev) {
        boolean result = false;
        switch (ev.getType()) {
        case ARRIVAL:
            // un carro puede llegar si hay espacio en cola
            result = (s.getQLength() < M - N - 1);
            break;
        case MIC_COMPLETION:
            // se puede terminar de tomar la orden si una persona esta haciendo
            // el pedido
            result = (s.getMicStatus() == ORDERING);
            break;
        default:
            // se puede terminar una orden si la persona correspondiente la esta
            // esperando
            if (ev.getPos() == N) {
                result = (s.getMicStatus() == COOKING);
            } else {
                result = (s.getStatus(ev.getPos()) == COOKING);
            }
        }
        return result;
    }

    /**
     * Computes the rate: the rate is lambda if an arraival occurs,
     * the rate is mu1 if a service type one is finished,
     * the rate is mu2 if an service type two is finished.
     *
     * @see SimpleMarkovProcess#rate(State, State, Event)
     */
    @Override
    public double rate(DriveThruState i, DriveThruState j, DriveThruEvent e) {
        switch (e.getType()) {
        case ARRIVAL:
            return lambda;
        case MIC_COMPLETION:
            return mu1;
        default:
            return mu2;
        }
    }

    /**
     * Computes the status of the destination when an event occurs
     *
     * @see SimpleMarkovProcess#dests(State, Event)
     */

    @Override
    public States<DriveThruState> dests(DriveThruState i, DriveThruEvent e) {
        int numServ = i.getAvlServs();
        CustStatus[] status = i.getStatus();
        CustStatus newMic = i.getMicStatus();
        int newQsize = i.getQLength();
        int numGone = 0;
        boolean micMoves = false;
        int k; // utility counter

        switch (e.getType()) {
        case ARRIVAL:
            if (i.getMicStatus() == EMPTY && numServ > 0) {

                newMic = ORDERING;
                numServ = numServ - 1;
            } else if (i.getMicStatus() == EMPTY && numServ == 0) {

                newMic = WAIT_MIC;
            } else if (i.getQLength() < M - N - 1) {

                newQsize = i.getQLength() + 1;
            }
            break;

```

```java
        case MIC_COMPLETION:
            newMic = COOKING;
            for (k = 0; ((k < N) && (status[k] != EMPTY)); k++)
                ;

            if (k != N) {
                status[k] = COOKING;
                newMic = EMPTY;
                micMoves = true;
            }
            break;

        default:
            numServ = numServ + 1;
            int p = e.getPos();
            if (p > 0 && p < N) {

                status[p] = BLOCKED_DONE;
            } else if (p == N) {
                newMic = BLOCKED_DONE;
            } else {

                status[0] = EMPTY;

                int pos1, pos2;

                for (k = 1; ((k < N) && status[k] == BLOCKED_DONE); k++)
                    ;
                numGone = k;
                if (k != N) {
                    pos1 = k;
                    pos2 = N - 1;
                    for (k = pos1; k <= pos2; k++) {
                        status[k - numGone] = status[k];
                    }
                }
                for (k = N - numGone; k < N; k++) {
                    status[k] = EMPTY;
                }
                if (newMic == COOKING) {
                    status[N - numGone] = newMic;
                    newMic = EMPTY;
                    micMoves = true;
                } else if (newMic == BLOCKED_DONE) {
                    newMic = EMPTY;
                    micMoves = true;
                }
            }
            break;
        } // end switch

        if (newMic == WAIT_MIC && numServ > 0) {
            newMic = ORDERING;
            numServ--;
        }
        if (micMoves) {
            if (i.getQLength() > 0 && numServ > 0) {
                newMic = ORDERING;
                numServ = numServ - 1;
                newQsize = i.getQLength() - 1;
            } else if (i.getQLength() > 0 && numServ == 0) {
                newMic = WAIT_MIC;
                newQsize = i.getQLength() - 1;
            }
        }
        StatesSet<DriveThruState> set = new StatesSet<DriveThruState>();
        set.add(new DriveThruState(status, newMic, newQsize, numServ));
        return set;
    } // end dests

    @Override
    public String description() {
        return "SISTEMA␣DRIVE␣THRU.␣" + "\nTasa␣de␣Entrada␣␣␣=␣" + lambda
                + "\nTasa␣en␣el␣Mic␣␣␣␣=␣" + mu1 + "\nTasa␣de␣sevicio␣2␣=␣"
                + mu2 + "\nPosicii¿½n␣del␣mic␣␣=␣" + N + "\nServidores␣␣␣␣␣␣␣␣=␣"
                + S + "\nCap␣en␣el␣sistema␣=␣" + M;
    }

    /**
     * Print all waiting times associated with each MOP
     */
    @Override
    public int printMOPs(PrintWriter out, int width, int decimals) {
        int namesWidth = super.printMOPs(out, width, decimals);
        // this rate work for all MOPs
        double ldaEff;
        try {
            ldaEff = getEventRate(ARRIVAL.ordinal());
            String[] names = getMOPNames();
            double waitTime;
            int N = names.length;
            namesWidth += 20;
            for (int i = 0; i < N; i++) {
                waitTime = 60 * getMOPsAvg(names[i]) / ldaEff;
                String name = "Waiting time for " + names[i];
                out.println(pad(name, namesWidth, false)
                        + pad(waitTime, width, decimals) + " minutes");
            }
        } catch (NotUnichainException e) {
            out.println(e);
        }
        return namesWidth;
    }

    /**
     * Main method.
     *
     * @param a
     *            Not used.
     */
    public static void main(String[] a) {
        // as in handout:
        DriveThru theDT = new DriveThru(80.0, 12.0, 30.0, 4, 2, 1);
        // DriveThru theDT = new DriveThru(80.0, 120.0, 30.0, 4, 2, 2);
        theDT.setDebugLevel(5);

        theDT.showGUI();
        theDT.printAll();
        theDT.printMOPs();
    }

} // class end

/**
 * This is a particular case of PropertiesState. Here, N is the position of the
 * microphone. The first N-1 components represent the status of the first queue, the
 * component N is the status of the microphone, the component N+1 is the number of clients in
 * the queue, and N+2 are the available servers.
 */
class DriveThruState extends State {

    // private int micPos;
    // private CustStatus micStatus;
    private int numQ;
    private int avlServ;
    private CustStatus[] prop = null;

    /**
     * This enumeration shows the different status for a customer.
     *
     */
    public enum CustStatus {
        /** Empty space. */
        EMPTY,
        /** In service. */
        ORDERING,
        /** A client in the microphone, but there are no servers available. */
        WAIT_MIC,
        /** The client order is being prepared. */
        COOKING,
        /** The order is ready but the client is blocked. */
        BLOCKED_DONE;
    }

    /**
     * Builds a State representing an empty system
     *
     * @param micPos
     * @param serv
     */
    DriveThruState(int micPos, int serv) {
        this(new CustStatus[micPos], EMPTY, 0, serv);
        for (int i = 0; i < prop.length; i++) {
            prop[i] = EMPTY;
        }
    }

    /**
     * Builds a DriveThru state.
     *
     * @param vec
     *            The states from the window until the microphone,
     *            without including the microphone.
     * @param mic
     *            Microphone status.
     * @param numQ
     *            Number of clients in the queue.
     * @param avServs
     *            Number of servers available.
     */

    DriveThruState(CustStatus[] statusVec, CustStatus micStatus, int numQ,
            int avServs) {
        prop = new CustStatus[statusVec.length + 1];
        int micPos = statusVec.length;
        System.arraycopy(statusVec, 0, prop, 0, micPos);
        prop[micPos] = micStatus;
        this.numQ = numQ;
        this.avlServ = avServs;
    }

    /**
     * Compute all the MOPs for this state
     */
    @Override
    public void computeMOPs(MarkovProcess mp) {
        int servEtapa1 = 0;
        int servEtapa2 = 0;
        int blockedDone = 0;
        int blockedBefore = 0;
        int total = 0;
        for (CustStatus s : prop) {
            servEtapa1 += (s == ORDERING) ? 1 : 0;
            servEtapa2 += (s == COOKING) ? 1 : 0;
            blockedDone += (s == BLOCKED_DONE) ? 1 : 0;
            blockedBefore += (s == WAIT_MIC) ? 1 : 0;
            total += (s != EMPTY) ? 1 : 0;
        }
        setMOP(mp, "Tamano Cola", getQLength());
        setMOP(mp, "Serv Ocupados Microfono ", servEtapa1);
        setMOP(mp, "Serv Ocupados Cocinando", servEtapa2);
        setMOP(mp, "Serv Ocupados ", servEtapa1 + servEtapa2);
        setMOP(mp, "Clientes Bloqueados antes de ordenar", blockedBefore);
        setMOP(mp, "Clientes Bloqueados con orden lista", blockedDone);
        setMOP(mp, "Clientes Bloqueados", blockedBefore + blockedDone);
        setMOP(mp, "Total clientes en Espera", blockedBefore + blockedDone
                + getQLength());
        setMOP(mp, "Total Clientes ", total + getQLength());
    }

    /**
     * Get the number of clients in the queue.
     *
     * @return Number of clients in the queue.
     */
    public int getQLength() {
        return numQ;
    }

    /**
     * Get the status of the of the i-th component.
     *
     * @param i
     *         index of the component
     *
     * @return Status of the i-th component.
     */
    public CustStatus getStatus(int i) {
        return prop[i];
    }

    /**
     * Get the vector of clients statuses.
     *
     * @return Status of components 0 to N-1.
     */
    public CustStatus[] getStatus() {
        int micPos = getMicPos();
        CustStatus[] status = new CustStatus[micPos];
        System.arraycopy(prop, 0, status, 0, micPos);
        return status;
    }

    /**
     * Get the status of the window.
     *
     * @return The status of the client at the microphone.
     */
    public CustStatus getMicStatus() {
        int n = prop.length - 1;
        return prop[n];
    }

    /**
     * Return the mic position.
     *
     * @return mic position index
     */
    public int getMicPos() {
        return prop.length - 1;
    }

    /**
     * Get the status of the window
     *
     * @return Status of the window.
     */
    public CustStatus getVentana() {
        return prop[0];
    }

    /**
     * Computes the number of available servers.
     *
     * @return Number of available servers.
     */
    public int getAvlServs() {
        return avlServ;
    }

    /**
     * @see jmarkov.basic.State#isConsistent()
     */
    @Override
    public boolean isConsistent() {
        // TODO Complete
        return true;
    }

    @Override
    public String label() {
        String stg = "";
        for (CustStatus s : prop) {
            switch (s) {
            case EMPTY:
                stg += "0";
                break;
            case ORDERING:
                stg += "m";
                break;
            case WAIT_MIC:
                stg += "w";
                break;
            case COOKING:
                stg += "c";
                break;
            case BLOCKED_DONE:
                stg += "b";
                break;
            }
        }
        return stg + "Q" + numQ;
        // return stg + "Q" + prop[micPos + 1] + "S" + prop[micPos + 2];
    }

    String statusDesc(CustStatus stat) {
        switch (stat) {
        case EMPTY:
            return "empty";
        case ORDERING:
            return "ordering,";
        case WAIT_MIC:
            return "waiting";
        case COOKING:
            return "cooking";
        default: // DONE
            return "blocked";
        }
    }

    /**
     * Describes the State
     *
     * @see jmarkov.basic.State#description()
     */
    @Override
    public String description() {
        String stg = "";
        int N = getMicPos();
        stg = "Queue CustStatus: (";
        for (int i = 0; i < N; i++) {
            stg += statusDesc(getStatus(i));
            stg += (i < N - 1) ? ", " : "";
        }
        stg += "). Mic status: " + statusDesc(getMicStatus());
        stg += ". Queue Size: " + getQLength();
        return stg;
    }

    /**
     * @see jmarkov.basic.State#compareTo(jmarkov.basic.State)
     */
    @Override
    public int compareTo(State j) {
        if (!(j instanceof DriveThruState))
            throw new IllegalArgumentException("Comparing wrong types!");
        DriveThruState u = (DriveThruState) j;
        int micPos = getMicPos();
        for (int k = 0; k <= micPos; k++) {
            if (getStatus(k).ordinal() > u.getStatus(k).ordinal())
                return +1;
            if (getStatus(k).ordinal() < u.getStatus(k).ordinal())
                return -1;
        }
        if (getQLength() > u.getQLength())
            return +1;
        if (getQLength() < u.getQLength())
            return -1;
        if (getAvlServs() > u.getAvlServs())
            return +1;
        if (getAvlServs() < u.getAvlServs())
            return -1;
        return 0;
    }

}

/**
 * This class implements the events in a Drive Thru.
 */
class DriveThruEvent extends jmarkov.basic.Event {
    /** Event types. */
    public static enum Type {
        /** Arrivale to the system. */
        ARRIVAL,
        /** Car at mic finishes service. */
        MIC_COMPLETION,
        /** Service completion for somebody who ordered. */
        SERVICE_COMPLETION;
    }

    private Type type; // event type
    private int position; // Position of the client whose order is complete

    /**
     * Creates an ARRIVAL or MIC_COMPLETION event.
     *
     * @param type
     */
    public DriveThruEvent(Type type) {
        assert (type == ARRIVAL || type == MIC_COMPLETION);
        this.type = type;
    }

    /**
     * Creates a Service Completion event at he given position.
     *
     * @param position
     *            Postion where the event occurs ( 0-based ).
     */
    public DriveThruEvent(int position) {
        this.type = SERVICE_COMPLETION;
        this.position = position;
    }

    /**
     * @return position where this event occurs. (valid only if type ==
     *         SERVICE_COMPLETION).
     */
    public int getPos() {
        assert (type == SERVICE_COMPLETION);
        return position;
    }

    /**
     * @return event type
     */
    public Type getType() {
        return type;
    }

    /**
     * @param micPos
     * @return A set with all the events in the system.
     */
    public static EventsSet<DriveThruEvent> getAllEvents(int micPos) {
        EventsSet<DriveThruEvent> eSet = new EventsSet<DriveThruEvent>();
        eSet.add(new DriveThruEvent(ARRIVAL));
        eSet.add(new DriveThruEvent(MIC_COMPLETION));
        for (int i = 0; i <= micPos; i++)
            eSet.add(new DriveThruEvent(i));
        return eSet;
    }

    @Override
    public String label() {
        String stg = "";
        switch (type) {
        case ARRIVAL:
            stg = "Arrival";
            break;
        case MIC_COMPLETION:
            stg = "MicEnd";
            break;
        default:
            stg = "SrvEnd(" + position + ")";
        }
        return stg;
    }

}
```

4.3.2 Results

Output for Drive Thru

```
SISTEMA DRIVE THRU.
Tasa de Entrada   = 80.0
Tasa en el Mic    = 120.0
Tasa de sevicio 2 = 30.0
Posici{\'o}n del mic  = 5
Servidores        = 4
Cap en el sistema = 14



System has 498 States.


MEASURES OF PERFORMANCE

NAME                                        MEAN      SDEV
Tamano Cola                                 4.503     2.693
Serv Ocupados Microfono                     0.550     0.498
Serv Ocupados Cocinando                     2.199     1.165
Serv Ocupados                               2.749     1.088
Clientes Bloqueados antes de ordenar        0.112     0.316
Clientes Bloqueados con orden lista         1.540     1.646
Clientes Bloqueados                         1.652     1.604
Total clientes en Espera                    6.155     3.487
Total Clientes                              8.903     3.396

EVENTS OCCURANCE RATES
NAME       MEAN RATE
Arrival       65.965
MicEnd        65.965
```

```
SrvEnd(0)     28.019
SrvEnd(1)      9.927
SrvEnd(2)      9.446
SrvEnd(3)      8.333
SrvEnd(4)      6.114
SrvEnd(5)      4.126

Tiempo de espera para Tamano Cola: 4.096 minutos
Tiempo de espera para Serv Ocupados Microfono : 0.5 minutos
Tiempo de espera para Serv Ocupados Cocinando: 2 minutos
Tiempo de espera para Serv Ocupados : 2.5 minutos
Tiempo de espera para Clientes Bloqueados antes de ordenar: 0.102 minutos
Tiempo de espera para Clientes Bloqueados con orden lista: 1.4 minutos
Tiempo de espera para Clientes Bloqueados: 1.503 minutos
Tiempo de espera para Total clientes en Espera: 5.598 minutos
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

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

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

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

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 α_i to reach the service phase i and with service rate μ_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 λ. 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 manged 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 $\mathcal{E} = \{a, c_1, c_2, \ldots, c_n\}$, 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 costumer could start in any of the n service phases, then the system could reach anyone of the first level n states with probability α_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 costumer 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 α_i. The Java code can be seen in Figure 7.

  – `rate (i,e)`: The rate of occurrence of event a is given simply by λ and the rate of occurrence of an event c_i is given by μ_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.

```
    public int getCurPH() {
        if (type == ARRIVAL)
            throw new IllegalArgumentException(
                    "Current phase is not defined for event " + ARRIVAL);
        return curPH;
    }

    /**
     * @return Returns the type.
     */
    public Type getType() {
        return type;
    }
```

Figure 6: `Active` method of class HiperExQueue.java

```
            // finish in phase n
            E.add(new HiperExQueueEvent(FINISH_SERVICE, n));
        }
        return E;
    }

    @Override
    public String label() {
        String stg = "";
        switch (type) {
        case ARRIVAL:
            stg = "Arrival";
            break;
        case FINISH_SERVICE:
            stg = "Ph(" + curPH + ")";
        }
        return stg;
    }
}

/**
 *  * This class define the states in the queue.
 * @author Julio Goez - German Riano. Universidad de los Andes.
 */
class HiperExQueueState extends PropertiesState {

    /**
     * We identify the states with the curPH of server in station, (1,
     * ..,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 λ_i, while services follow a PH distribution with parameters $(\boldsymbol{\alpha}^{(i)}, \mathbf{A}^{(i)})$. 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

```java
    /**
     * Returns the service phase of process
     * @return Service phase
     */
    public int getSrvPhase() {
        return this.prop[0];
    }

    /**
     * @see jmarkov.basic.State#isConsistent()
     */
    @Override
    public boolean isConsistent() {
        // TODO Complete
        return true;
    }

    /**
     * Returns the service status
     * @return Service status (1 = busy, 0 = free)
     */
    public int getSrvStatus() {
        return (getSrvPhase() == 0) ? 0 : 1;
    }

    @Override
    public HiperExQueueState clone() {
        return new HiperExQueueState(getSrvPhase());
    }
```

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

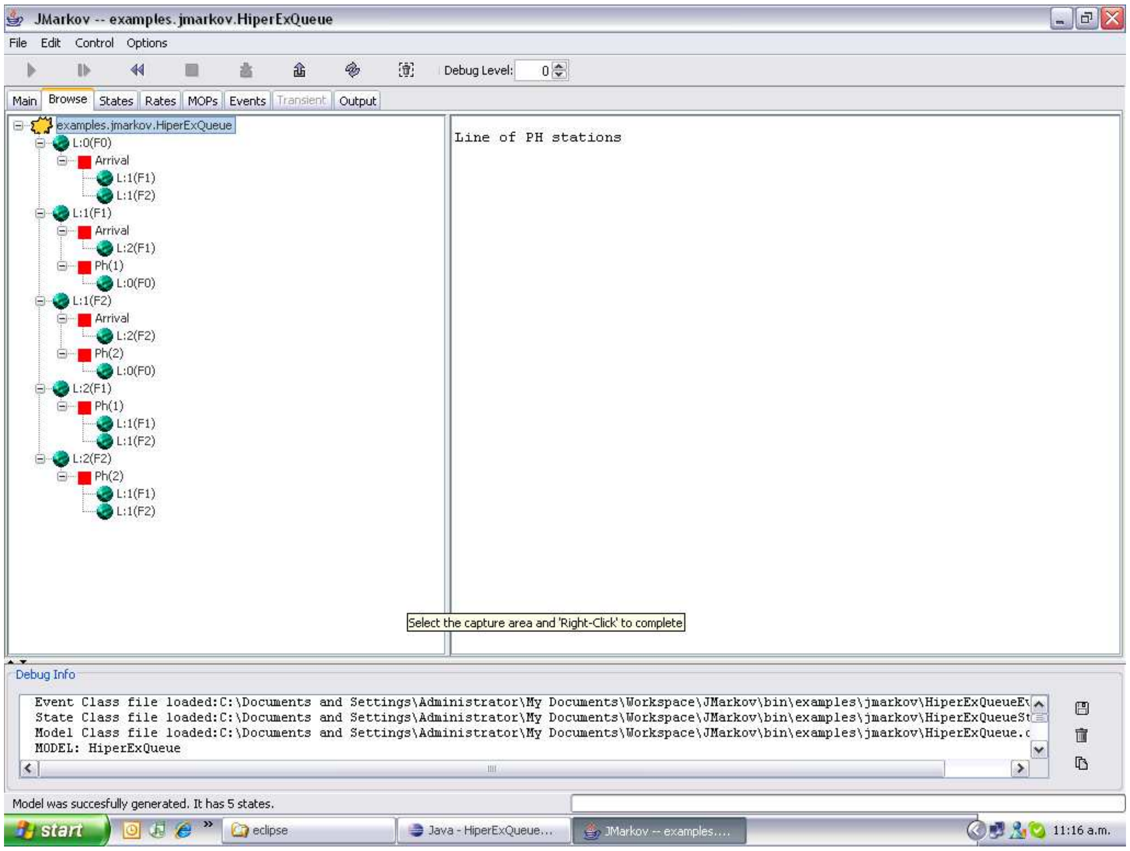
where the *level* holds the number of low-priority jobs, while all other information necessary to describe the system state is left for the *phase*. The phase thus holds the number of high-priority jobs in the system and the service phase of the job currently in service. We also include in the phase 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.

```java
class PriorityQueueMPHPHPreemptState extends PropertiesState {
        public PriorityQueueMPHPHPreemptState(int numberHiJobs, int servicePhase, int serviceType) {
                super(3);
                setProperty(0, numberHiJobs);
                setProperty(1, servicePhase);
                setProperty(2, serviceType);
        }
}
```

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.

```java
class PriorityQueueMPHPHPreemptEvent extends Event {
        public enum Type {
                ARRIVAL_HI,
                SERVICE_END_HI,
                SERVICE_PHASECHG_HI,
                ARRIVAL_LOW,
                SERVICE_END_LOW,
                SERVICE_PHASECHG_LOW
        }
        Type eventType;
        int eventPhase;
```


JMarkov -- examples.jmarkov.HiperExQueue
File Edit Control Options
Debug Level: 0
Main Browse States Rates MOPs Events Transient Output
Select the capture area and 'Right-Click' to complete
Debug Info
start
eclipse
Java - HiperExQueue...
JMarkov -- examples....
11:16 a.m.

```
examples.jmarkov.HiperExQueue
  L:0(F0)
    Arrival
      L:1(F1)
      L:1(F2)
  L:1(F1)
    Arrival
      L:2(F1)
    Ph(1)
      L:0(F0)
  L:1(F2)
    Arrival
      L:2(F2)
    Ph(2)
      L:0(F0)
  L:2(F1)
    Ph(1)
      L:1(F1)
      L:1(F2)
  L:2(F2)
    Ph(2)
      L:1(F1)
      L:1(F2)
```

```
Line of PH stations
```

```
Event Class file loaded:C:\Documents and Settings\Administrator\My Documents\Workspace\JMarkov\bin\examples\jmarkov\HiperExQueueEv
State Class file loaded:C:\Documents and Settings\Administrator\My Documents\Workspace\JMarkov\bin\examples\jmarkov\HiperExQueueSt
Model Class file loaded:C:\Documents and Settings\Administrator\My Documents\Workspace\JMarkov\bin\examples\jmarkov\HiperExQueue.c
MODEL: HiperExQueue
```

Model was succesfully generated. It has 5 states.

Figure 9: GUI example of jMarkov

```
MEASURES OF PERFORMANCE

NAME                          MEAN        SDEV

Expected Level             0.14286          ?
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 `PriorityQueueMPHPHPreempt`, which, as shown in the following snippet, extends the `GeomProcess` class since our model is a QBD.

```
1 public class PriorityQueueMPHPHPreempt extends
2         GeomProcess<PriorityQueueMPHPHPreemptState, PriorityQueueMPHPHPreemptEvent>{
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;
```

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;
```

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         PriorityQueueMPHPHPreempt model = new PriorityQueueMPHPHPreempt(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
```

```
NAME                              MEAN       SDEV
Expected Level                    6.47779
Number High Jobs                  1.02821    1.79758
High Jobs Blocking Probability    0.00494    0.07010
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.

Index

`active`, 3
`dests`, 3
`rate`, 3

CTMC, 2

DTMC, 2

Generator Matrix, 2

Markov Decision Processes, 2
MDP, 2

Object Oriented Programming, 4
OOP, 4

PH, 2
Phase-Type Distributions, 2

QBD, 2
Quasi-Birth and Death Processes, 2