Algorithm xxx: jMarkov: an Integrated Framework for Markov Chain Modeling

JUAN F. PÉREZ, University of Melbourne
DANIEL F. SILVA, Auburn University
JULIO C. GÓEZ, NHH Norwegian School of Economics
ANDRÉS SARMIENTO, ANDRÉS SARMIENTO-ROMERO, Universidad de los Andes
RAHA AKHAVAN-TABATABAEI, Universidad de los Andes and Sabanci University
GERMÁN RIAÑO, Universidad de los Andes

Markov chains (MC) are a powerful tool for modeling complex stochastic systems. Whereas a number of tools exist for *solving* different types of MC models, the first step in MC modeling is to define the model parameters. This step is however error prone and far from trivial when modeling complex systems. In this article we introduce jMarkov, a framework for MC modeling that provides the user with the ability to define MC models from the basic rules underlying the system dynamics. From these rules, jMarkov automatically obtains the MC parameters and solves the model to determine steady-state and transient performance measures. The jMarkov framework is composed of four modules: (i) the main module supports MC models with a finite state space; (ii) the jQBD module enables the modeling of Quasi-Birth-and-Death processes, a class of MCs with infinite state space; (iii) the jMDP module offers the capabilities to determine optimal decision rules based on Markov Decision Processes; and (iv) the jPhase module supports the manipulation and inclusion of phase-type variables to represent more general behaviors than that of the standard exponential distribution. In addition, jMarkov is highly extensible, allowing the users to introduce new modeling abstractions and solvers.

Categories and Subject Descriptors: G.3 [**Probability and Statistics**]: Markov processes; G.4 [**Mathematical software**]: Documentation; I.6 [**Simulation and Modeling**]: Model Development

CCS Concepts: •**Mathematics of computing** → **Markov processes; Solvers;** •**Computing methodologies** → **Modeling methodologies;**

General Terms: Design, Documentation

Additional Key Words and Phrases: Stochastic modeling, Markov chains, Quasi-birth-and-death processes, Phase-type distributions, Markov decision processes

ACM Reference Format:
Juan F. Pérez, Daniel F. Silva, Julio C. Góez, Andrés Sarmiento, Andrés Sarmiento-Romero, Raha Akhavan-Tabatabaei and Germán Riaño, 2016. Algorithm xxx: jMarkov: an Integrated Framework for Markov Chain Modeling. *ACM Trans. Math. Softw.* V, N, Article 0000 (2015), 22 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Stochastic modeling has become an important tool to analyze complex systems subject to uncertainty, which are prevalent in many areas, including manufacturing, logistics, computer and telecommunications networks, among others. A stochastic model can be used to perform what-if analyses to understand the effect of critical design parameters on overall system metrics, such as performance or reliability. Within the set of stochastic models, Markov chains (MC) have received particular attention due to their modeling flexibility, analytical tractability, and amenability for numerical analysis.

For these reasons, a number of software tools have been proposed for building and solving MC models. In fact, most of these tools focus on providing numerical methods to *solve* the model to obtain the MC stationary measure, from which relevant steady-state metrics can be computed. However, before solving the model, one first needs to determine the model parameters, such as the MC generator matrix, which must be provided as input to the solvers, but this task is far from trivial when modeling complex systems. In this paper, we introduce jMarkov, an extensible object-oriented tool for the modeling and solution of MC models. The key distinguishing feature of jMarkov lies in its *modeling*

The research of Juan F. Pérez is supported by the ARC Centre of Excellence for Mathematical and Statistical Frontiers (ACEMS).
Author's addresses: J. F. Pérez, School of Mathematics and Statistics, University of Melbourne, Melbourne, Australia; D. F. Silva, Department of Industrial and Systems Engineering, Auburn University, Auburn, AL, USA; J. C. Góez, Department of Business and Management Science, NHH Norwegian School of Economics, Bergen, Norway; R. Akhavan-Tabatabaei, (current address) School of Management, Sabanci University, Istanbul, Turkey; A. Sarmiento, A. Sarmiento-Romero and G. Riaño, Department of Industrial Engineering, Universidad de los Andes, Bogotá, Colombia.

DOI: http://dx.doi.org/10.1145/0000000.0000000

capabilities, where the user needs only to define the rules that govern the system dynamics. Based on these rules, jMarkov *builds the model parameters*, solves the model, and computes the measures of performance.

Furthermore, jMarkov not only supports finite MCs, but it is unique as a modeling tool for Quasi-Birth-and-Death processes (QBD) and Markov Decision Processes (MDP). QBDs allow the user to consider models with an infinite state space, whereas MDPs support the computation of optimal decision policies. While a number of tools provide *solvers* for these models, jMarkov takes on the *modeling* step, for which no alternative tools exist. Thus, the user can offload to jMarkov the cumbersome and error-prone procedure of building the model parameters. In addition, the framework is complemented by a module to manipulate phase-type (PH) distributions, a class of distributions that can be incorporated in MC models to consider behaviors more general than those allowed by the standard exponential distribution. An early version of some of these modules was described in [Riaño and Góez 2006; Pérez and Riaño 2006].

A key feature in the design of jMarkov is the separation of modeling from solving, allowing the analyst to concentrate on describing the system without needing to worry about the implementation of the solution algorithms. In this regard, the jMarkov framework resembles software packages for mathematical programming that offer Java libraries, like Gurobi [Gurobi Optimizaton 2014] and CPLEX [CPLEX Optimizer 2015], which the user employs to specify an optimization problem, without needing to know the details of the underlying solution algorithms. Yet, the platform is flexible enough to allow new solvers to be implemented and added by third-party developers.

We developed the jMarkov framework in Java [Sun Microsystems 2006], and designed it to make extensive use of the abstraction principle that Java is built on. Java makes a great choice to enable the extensibility of jMarkov, thanks to its object-oriented nature. In addition, Java is a widely-used, platform independent, and well-supported language, allowing a large audience to use jMarkov without the need to learn a new programming language. Also, the extensibility of jMarkov is further supported by its open-source release and the availability of many examples (currently over 50) and detailed documentation. jMarkov is publicly available as a project of the Computational Infrastructure for Operations Research (COIN-OR). Source code, documentation, examples, and case studies are available at [jMarkov website 2016].

The following sections present the four main modules, jMarkov, jQBD, jMDP, and jPhase, followed by examples, and a description of extensions and applications.

2. THE JMARKOV MODULE

In this section we describe the main features of the jMarkov core module, which focuses on finite MC models. After a short review of MCs, we describe in detail the state-space search algorithm that enables the automatic model-building feature of jMarkov. We explain how jMarkov allows a user to exploit this algorithm by implementing a few classes and methods to describe the system state and the events that modify the state. To this end, we also illustrate the main classes in jMarkov that a user will typically rely on to build MC models. This description also serves as a stepping stone for the extensions introduced in sections 3 and 4 to handle QBD and MDP models.

2.1. Background

A homogeneous Continuous Time Markov Chain (CTMC) [Kulkarni 1995] is a stochastic process $\{X(t), t \geq 0\}$, defined on a countable state space $\mathcal{S}$, that satisfies the Markov property. The jMarkov module focuses on irreducible CTMCs with finite $\mathcal{S}$, and the following discussion assumes this condition. A CTMC with $n = |S|$ states is completely determined by two parameters: i) its initial distribution (row) vector $\boldsymbol{\alpha}$, with entries $\alpha_i = \mathbb{P}\{X(0) = i\}$ for $i \in \mathcal{S}$, i.e, the probability that the system is in state i at time 0; and ii) its generator matrix $\boldsymbol{Q} \in \mathbb{R}^{n \times n}$, with non-positive diagonal components such that $-q_{ii}$ is the exponential holding rate for state i, whereas the off-diagonal elements q_{ij} represent the transition rate from state i to state j. Since $q_{ii} = \sum_{j=1, j \neq i}^{n} q_{ij}$, one can equivalently define the MC by relying on the rate matrix $\mathbf{T}$, which has the same entries as $\boldsymbol{Q}$, except for its diagonal elements, which are zero. In fact, the objective of the state-space search algorithm implemented by jMarkov is to determine the matrix $\mathbf{T}$.

From $\boldsymbol{\alpha}$ and $\boldsymbol{Q}$, it is possible to describe the transient behavior of the CTMC, which is given by the matrix $\mathbf{P}(t) \in \mathbb{R}^{n \times n}$ with entries $p_{ij}(t) = \mathbb{P}\{X(t+s) = j | X(s) = i\}$, $i, j \in \mathcal{S}$, i.e., the probability that the system is in state j at time t given that is started in state i at time 0. This matrix can be computed as $\mathbf{P}(t) = e^{\boldsymbol{Q}t}$ for $t \geq 0$. In addition, it is also of interest to determine the stationary distribution of the process, i.e., the long-term fraction of time that the process spends in state i, which we label π_i.

The stationary distribution $\boldsymbol{\pi} = [\pi_1, \pi_2, \dots,]$ of an irreducible CTMC is obtained as the solution to the system of equations $\boldsymbol{\pi Q} = \mathbf{0}$, $\boldsymbol{\pi 1} = 1$, where $\mathbf{1} \in \mathbb{R}^{n \times 1}$ is a column vector of ones.

2.2. Algorithms

2.2.1. Building Large-scale Markov Chains. Transitions in a CTMC are triggered by the occurrence of events, for instance arrivals and departures in a queueing model. The matrix $\mathbf{T}$ can be decomposed as $\mathbf{T} = \sum_{e \in \mathcal{E}} \mathbf{T}^{(e)}$, where $\mathbf{T}^{(e)}$ 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 all the reachable states in the model and the transitions among them. However, it is possible to determine what events occur in each state, and the destination states produced by each transition when it occurs, thus obtaining the $\mathbf{T}^{(e)}$ matrices that compose $\mathbf{T}$. jMarkov works based on this observation, using the *BuildRS* algorithm [Ciardo 2000], which builds the state space and the rate matrix $\mathbf{T}$ by a deep exploration of the state graph. Algorithm 1 illustrates the *BuildRS* algorithm, which starts from a single state i_0 and the set of events $\mathcal{E}$ to determine the full state space $\mathcal{S}$ and the rate matrix $\mathbf{T}$. The algorithm goes through each state not yet analyzed (set $\mathcal{U}$), and determines whether each event $e \in \mathcal{E}$ is active or not in that state, according to the `active` method. If an event is active, the next step is to determine the set $\mathcal{D}$ of all reachable states when this event occurs, by relying on the `dests` method. For each state in $\mathcal{D}$ the algorithm then uses the method `rate` to determine the associated transition rate, and adds all newly found states to the set $\mathcal{U}$ for further analysis. The algorithm stops when no states remain to be analyzed in $\mathcal{U}$. This algorithm therefore requires the definition of states, events, and of the methods `active`, `dests`, and `rate`. These are precisely the definitions that jMarkov requires the user to code as Java classes and methods. Once these basic methods are defined, jMarkov builds the matrix $\mathbf{T}$, from which it derives the matrix $\boldsymbol{Q}$ and computes the stationary distribution and the associated measures of performance. Notice that the *BuildRS* algorithm underlying jMarkov slightly differs from the one in [Ciardo 2000] in that each event can trigger transitions to multiple states instead of a single one, as captured by the set $\mathcal{D}$ returned by the method `dests`.

2.2.2. Measures of Performance. When studying MCs, the analyst is usually interested in the transient and steady-state behavior of a set 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 $i \in \mathcal{S}$. 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 thus $E\big(r(X(t))\big) = \boldsymbol{\alpha}\mathbf{P}(t)\mathbf{r}$. Similarly, for an irreducible CTMC, the long run rate at which the system receives rewards is given by

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

jMarkov provides mechanisms to define this type of rewards and can compute both transient and steady-state MOPs. There are other types of rewards, like the expected time in the system, which are computed using Little's law [Little 1961]. As these results rely on the CTMC being irreducible, jMarkov throws an exception of the type `NotUniChainException` in case the resulting CTMC is reducible.

2.3. Architecture

To better describe the capabilities of jMarkov, this section provides some details of its architecture, which consists of three main packages: Build, Basic, and Solvers.

2.3.1. The Build Package. The Build package plays a central role in jMarkov since it contains the classes that take care of building the state space and transition matrices. As depicted in Table I, the main classes are `MarkovProcess`, `SimpleMarkovProcess`, and `GeomProcess`. The class `MarkovProcess` implements the *BuildRS* algorithm described in the previous section, while `SimpleMarkovProcess` extends this class and defines the abstract methods `active`, `dests`, and `rate`. A typical user will therefore extend the `SimpleMarkovProcess` class and implement these three abstract methods. Alternatively, the user could directly extend the `MarkovProcess` class, which requires the implementation of the `activeTransitions` method. This method combines the `dests` and `rate` methods mentioned before, determining at once, for each state and event, the set of destination states and the associated rates. This implementation thus avoids traversing the set $\mathcal{D}$ in Algorithm 1 twice, first to determine the destination states and then to determine the transition rates. While more efficient, this implementation may also require some previous experience with the jMarkov modeling approach, and we therefore expect users to extend the `SimpleMarkovProcess` class, implementing the `active`, `dests`, and `rate` methods separately. Both `SimpleMarkovProcess` and `MarkovProcess` contain tools to communicate with the

ALGORITHM 1: BuildRS algorithm

```
Input: 𝒰 = {i0}, ℰ                          /* Input: the initial state and the set of events */
Output: 𝒮, T                                /* Outputs: the state space and the entries of T */
𝒮 = ∅;
while 𝒰 ≠ ∅ do
    Choose an i ∈ 𝒰;
    𝒮 := 𝒮 ∪ {i};
    𝒰 := 𝒰\{i};
    foreach e ∈ ℰ do
        if active (i, e) then               /* Determine if the event e may occur while in state i */
            𝒟 :=dests (i, e);
            foreach j ∈ 𝒟 do                /* Check all reachable states found by dests */
                T_ij := T_ij+rate (i, j, e);
                if j ∉ 𝒮 ∪ 𝒰 then            /* If j is unexplored add it to 𝒰 */
                    𝒰 := 𝒰 ∪ {j};
                end
            end
        end
    end
end
```

Table I. jMarkov Main Packages and Classes

Build Package	Basic Package	Solvers Package
MarkovProcess	Event	SteadyStateSolver
SimpleMarkovProcess	State	TransientSolver
GeomMarkovProcess	PropertiesEvent	GeomSolver
	PropertiesState	JamaSolver
	GeomState	Mtjsolver
	GeomRelState	MtjLogRedsolver

solvers to compute steady state and transient solutions, and output them in different formats. Finally, the `GeomProcess` class extends the `MarkovProcess` class to model QBD processes, which we describe in Section 3.

2.3.2. The Basic Package. In addition to the `active`, `dests`, and `rate` methods, the other key step in setting up an MC model in jMarkov is the definition of events and states. To this end, the Basic package contains the abstract classes `State` and `Event`, which the user can extend to code the description of the states and events, respectively. The user has the freedom to choose the coding that best describes the states in a model, like any combination of integers, strings, etc. The only requirement here is to establish a complete ordering among the elements since, for efficiency, jMarkov works with ordered sets. For simplicity, however, this package also provides the `PropertiesState` class to describe the state as an array of integers, which is appropriate for many applications. Similarly, the `PropertiesEvent` class provides a simple event description in terms of integers. The two other classes of this package, `GeomState` and `GeomRelState`, are described in Section 3 as part of the `jQBD` module.

2.3.3. The Solvers Package. As we mentioned in the introduction, jMarkov separates modeling from solving. Once the model is built using the classes in the `Build` and `Basic` packages, it is solved with one of the solvers in the `Solver` package. As depicted in Table I, there are three (abstract) solvers, namely, `SteadyStateSolver`, `TransientSolver`, and `GeomSolver`. As the names indicate, the first two compute steady state and transient probabilities for finite MCs, whereas the latter is used for QBDs. The remaining classes listed in Table I extend either of these abstract classes to provide ready-to-use solvers. These solvers rely on the Java packages JAMA [Hicklin et al. 2005] and MTJ [Heimsund 2005], thus exploiting their capabilities to handle matrix operations for dense and sparse matrices, as well as a number of methods to solve the linear systems underlying the solution of the MC model. Notice that the user can also implement other solvers, e.g., making use of other libraries or developing a solver for a specific class of MC models. Alternatively, it is possible to export the obtained model and employ solvers available in other tools. This can be achieved for instance by using standard matrix exchange formats, or by directly calling the jMarkov libraries from other environments such as MATLAB. These features, not present in other MC modeling packages, highlight the flexibility of jMarkov.

2.3.4. The Graphical User Interface. Another feature of jMarkov is its graphical user interface (GUI). This GUI presents the measures of performance, displays debugging messages, provides a full list of the states found in the state generation process, as well as a spreadsheet-like view of the rate matrix T. This information can be very valuable to the user when developing a new model, as it allows for a detailed view of the model parameters. For instance, the user can verify, for a small-scale example, that all the transition rates are as expected, or that the transitions obtained for a subset of states, and their associated rates, comply with the high-level definition. The GUI thus supports the user in the verification of the model correctness.

2.4. Related Work

There are other academic software packages similar to jMarkov. MARCA [Stewart 1996] is a Fortran 77 package that supports modeling large MCs and provides a large number of numerical solution procedures. One limitation of MARCA is that, to make use of its model-building capabilities, it requires the user to define the state as a set of integers and to provide a description of the transitions in terms of buckets. This implies determining, for each entry in the state descriptor (bucket), the destination buckets when a transition occurs. While this description may be simple to obtain for certain models, it may be difficult or unfeasible to provide such description for a complex model. Another package is SMART [Ciardo et al. 2002], which provides a modeling language with its own syntax for the definition of the MC. This requires the user to learn a new language to make use of SMART. SMART also offers built-in solvers and supports both continuous and discrete time MCs. Another issue, shared by both MARCA and SMART, is that they are no longer supported, making the tools and their documentation hard to access. Another tool is DNAmaca [Knottenbelt 1996], which also defines a modeling language and a parser that writes the corresponding C++ code and compiles the classes to obtain the state space generator for the model. This generator feeds the tools to compute the steady state probabilities and measures of performance. The DNAmaca language is actually oriented towards Petri net analysis, and DNAmaca is now embedded within the Java Platform Independent Petri net Editor (PIPE) [Dingle et al. 2009]. As SMART, DNAmaca requires the user to use a specific language to describe the model. Also, it does not provide transient analysis. In addition to the above, we must point out that all these tools offer a rigid combination of modeling and solution, without easy means to add new solvers or alternative state-space search method. In contrast, jMarkov is an easily extensible tool, with an architecture that readily allows for novel solvers or other state-space search methods. Also, jMarkov is an open-source project, with easy to access source code and documentation, and it is implemented in Java, a widely used programming language. Moreover, the state definition in jMarkov is fairly general, allowing for descriptors closer to the actual system under analysis. Finally, as we shall see in the next sections, jMarkov is not limited to modeling finite MCs. It also supports QBD and MDP models, which are not currently supported by any other tool.

There are also commercial software packages that include MC modules like RAM Commander [Advanced Logistic Department 2013] and ITEM Toolkit, [ITEM Software (USA) Inc. 2011], which provide graphical interfaces to build MCs and find the steady state probabilities. However, these tools are designed to graphically build models with a few states and not to build and solve large MCs. These graphical interfaces work well to manually build small models, but they do not provide support to automatically build large-scale models.

3. THE JQBD MODULE

While finite MC models offer great flexibility, in many applications it is natural to assume that some variables can take a countably infinite number of values. This is the case for instance in queueing applications where the queue-length is assumed to be countably infinite. To model such systems, jMarkov offers support, via the `jQBD` module, for Quasi-Birth-and-Death (QBD) processes [Latouche and Ramaswami 1999], which are MCs with an infinite state space, but with a very specific structure that makes them numerically tractable. Notice that, due to their infinite state space, the *BuildRS* algorithm introduced in Section 2 cannot be directly applied as it would never terminate. In this section we introduce QBDs, and explain how the *BuildRS* algorithm can be modified to enable jMarkov to model them.

3.1. Background

Consider a MC $\{X(t) : t \geq 0\}$ 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 queueing applications, the level usually represents the number of items in the system, whereas the phase may represent the stages of a service process. While

in a birth-and-death process the only allowed transitions are to the two adjacent states [Kulkarni 1995], in a QBD a transition is allowed to any state in the same or in adjacent *levels*. Also, for $n \geq 1$ the transition rates are independent of the level n. Therefore, the generator matrix $\boldsymbol{Q}$ has the following structure

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{B}_{00} & \boldsymbol{B}_{01} & & & \\ \boldsymbol{B}_{10} & \boldsymbol{A}_1 & \boldsymbol{A}_0 & & \\ & \boldsymbol{A}_2 & \boldsymbol{A}_1 & \boldsymbol{A}_0 & \\ & & \ddots & \ddots & \ddots \end{bmatrix}, \tag{1}$$

where, as usual, the rows add up to 0. In general, the level zero might have a number of phases $m_0 \neq m$. Note that the matrix $\boldsymbol{B}_{00}$ has size $m_0 \times m_0$, whereas $\boldsymbol{B}_{01}$ and $\boldsymbol{B}_{10}$ are matrices of sizes $(m_0 \times m)$ and $(m \times m_0)$, respectively. Assuming the QBD is ergodic, there exists a steady state distribution $\boldsymbol{\pi}$ that is the unique solution $\boldsymbol{\pi}$ to the system $\boldsymbol{\pi Q} = \mathbf{0}$, $\boldsymbol{\pi 1} = 1$. Partitioning the vector $\boldsymbol{\pi}$ by levels $\boldsymbol{\pi} = [\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \ldots]$, these sub-vectors satisfy $\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \mathbf{R}$, $n > 1$. Here $\mathbf{R}$ is a square matrix of order m, called the rate matrix [Neuts 1981], that is the minimal non-negative solution to the equation $\boldsymbol{A}_0 + \mathbf{R}\boldsymbol{A}_1 + \mathbf{R}^2\boldsymbol{A}_2 = \mathbf{0}$.

3.2. Algorithms

3.2.1. State-space Search for QBD processes. As with finite MCs, QBDs can be modeled with the `jQBD` module by defining states and events, and describing the system dynamics via the `active`, `dests`, and `rate` methods. However, due to the infinite state space of the QBD, we cannot rely on the *BuildRS* algorithm directly. To handle QBDs we therefore extend the *BuildRS* algorithm via two key modifications. First, we modify the `dests` method to incorporate the concept of level that defines a QBD. Since from a state (n, i), transitions are allowed only to states in levels $n-1$, n, and $n+1$, the `dests` method determines the destination state as a *relative* state $(-1, j)$, $(0, j)$, $(1, j)$. This change requires replacing the `State` class for finite MCs by two classes: `GeomState` for absolute states that belong to a certain level $n \geq 0$; and `GeomRelState` for relative states that indicate whether in the destination state the level increases, decreases or remains the same. Second, rather than searching the whole state space, we limit the search to the first three levels of the generator matrix (1), from which we can extract the matrices $\boldsymbol{B}_{00}$, $\boldsymbol{B}_{01}$, $\boldsymbol{B}_{10}$, $\boldsymbol{A}_0$, $\boldsymbol{A}_1$, and $\boldsymbol{A}_2$. These modifications are used by the `GeomProcess` class to override the *BuildRS* algorithm and obtain the parameters of the QBD, which it passes to the solver to compute the matrix $\mathbf{R}$, as described in the next section.

3.2.2. Computing the rate matrix. A number of iterative algorithms have been proposed to compute the matrix $\mathbf{R}$. jQBD provides an implementation of the logarithmic-reduction algorithm [Latouche and Ramaswami 1999] in the class `MtjLogRedSolver`. We choose this algorithm as it is the only one, together with cyclic-reduction [Bini et al. 2005], to offer a quadratic convergence rate, a feature particularly beneficial for models that are close to null-recurrent but stable, as in queues with a very high utilization.

Once the rate matrix $\mathbf{R}$ has been found, the $\boldsymbol{\pi}_0$ and $\boldsymbol{\pi}_1$ vectors are determined by solving the following linear system of equations

$$[\boldsymbol{\pi}_0 \;\; \boldsymbol{\pi}_1] \begin{bmatrix} \boldsymbol{B}_{00} & \boldsymbol{B}_{01} \\ \boldsymbol{B}_{10} & \boldsymbol{A}_1 + \mathbf{R}\boldsymbol{A}_2 \end{bmatrix} = [\mathbf{0} \;\; \mathbf{0}], \quad \boldsymbol{\pi}_0 \mathbf{1} + \boldsymbol{\pi}_1 (\mathbf{I} - \mathbf{R})^{-1} \mathbf{1} = 1. \tag{2}$$

3.2.3. Measures of performance for QBDs. We consider two types of MOPs 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 $n \geq 1$. The long-run value for such a measure of performance is computed as $\sum_{n=1}^{\infty} \boldsymbol{\pi}_n \mathbf{r} = \boldsymbol{\pi}_1 (\mathbf{I} - \mathbf{R})^{-1} \mathbf{r}$, where $\mathbf{r}$ is a size-m column vector with components r_i. The second type of reward has the form $n r_i$ whenever the system is in phase i of level n. Its long-run value is $\sum_{n=1}^{\infty} n \boldsymbol{\pi}_n \mathbf{r} = \boldsymbol{\pi}_1 \mathbf{R} (\mathbf{I} - \mathbf{R})^{-2} \mathbf{r}$. The first type of performance measure can be used, for instance, to determine the steady-state utilization of a server whose state is described through the phase variable. A common example of the second type of performance measure is the mean queue length, where the queue length is modeled as the level variable. We conclude this section by highlighting that, thanks to its close integration with jMarkov, models built with `jQBD` can also exploit the GUI described in Section 2.3.4 to debug and verify the correctness of the model.

3.3. Related Work

Whereas a number of tools exist to solve QBDs, the modeling capabilities of jMarkov are not available in any other tool. Probably the most well-known tool to analyze QBDs is SMCSolver [Bini et al. 2009], a software with versions in both Fortran 90 and Matlab, where the former also offers a graphi-

cal user interface. It contains state-of-the-art algorithms for solving QBDs, as well as other structured MCs (GI/M/1 and M/G/1-types). Similarly, MAMSolver [Riska and Smirni 2007] offers C++ and Matlab implementations of methods to solve QBD, M/G/1 and G/M/1-type Markov chains. In particular, it implements the ETAQA method [Riska and Smirni 2007] to efficiently compute average metrics such as the mean queue length when modeling queueing systems. Also, Butools [Butools 2016] offers methods to analyze QBDs and M/G/1-type Markov chains, as well as Markovian fluid models, implemented in Matlab and Python. While jMarkov offers support for QBDs only, its distinctive feature lies in its modeling capabilities, which none of the other tools have. In fact, existing tools focus on *solving* the model, assuming as input the matrices $\boldsymbol{A}_0$, $\boldsymbol{A}_1$, and $\boldsymbol{A}_0$, and the other submatrices that define the generator $\boldsymbol{Q}$ in (1). jMarkov is the only one with *modeling* capabilities, where these matrices are built from first principles.

4. THE JMDP MODULE

Markov Decision Processes (MDP) [Puterman 1994] enable the modeling and solution of complex multi-stage decision problems in the presence of uncertainty. MDPs build on the theory of Markov chains to model the random system dynamics, and incorporate a decision-maker that takes actions at each stage, affecting both the system evolution and the associated rewards or penalties. The jMarkov framework includes the `jMDP` module for modeling and solving MDPs. While there are several tools available for *solving* discrete-time MDPs, `jMDP` is unique in its capabilities for *modeling* MDPs. This includes support for modeling problems with or without events and for modeling both discrete-time and continuous-time problems. The next sections provide a short introduction to MDPs and describe in detail how the `jMDP` module extends the state-space search methods introduced in Section 2 to automatically build MDP models. We also describe the solution methods provided by `jMDP` to analyze the different types of MDP models supported. An example of a `jMDP` model is provided in Section 7.

4.1. Background

The problems that can be modeled and solved with the `jMDP` module can be classified in two categories based on the planning horizon: finite and infinite horizon problems. Infinite horizon problems can be in either discrete or continuous time. For each model type in these categories, `jMDP` provides a Java class for its modeling. Additionally, we provide classes to model MDP problems with events, and actions that depend on events, a technique that results in a more natural representation. In this section we present detailed definitions for discrete time, infinite horizon problems, whereas other problem types are described more briefly.

4.1.1. Infinite Horizon Discrete Time MDP Problems (DTMDP). Consider a discrete-space, discrete-time, bivariate random process $\{(X_n, A_n), n = 0, 1, \ldots\}$. $X_n \in \mathcal{S}$ represents the state of the system at period n, and $A_n \in \mathcal{A}$ is the action taken at that period. The sets $\mathcal{S}$ and $\mathcal{A}$ are the spaces of all states and actions respectively, and we assume that both are finite. We also assume that all system parameters are stationary. The system state evolves according to the actions taken at each period, which leads the system to another state according to a probability distribution. We define $p_{ij}(a)$ as the probability of transitioning to state j whenever action a is taken and the system is in state i. A *deterministic policy* is a function d that assigns an action from the set $\mathcal{A}$, for each state. For each action a taken from state i, a finite cost $c(i, a)$ is incurred. Consequently it is possible to define a total expected cost $v^d(i)$ incurred when following policy d, called the *value function*, as

$$v^d(i) = \mathbb{E}_d \left[\sum_{n=0}^{\infty} c(X_n, A_n) \Big| X_0 = i \right], \quad i \in \mathcal{S}, \tag{3}$$

where $\mathbb{E}_d$ is the expectation operator over the probability distribution associated with policy d.

The objective is to find a policy $d^* \in D$, where D is the set of all deterministic policies, that minimizes the objective function shown above, $v^*(i) = \inf_{d \in D} v^d(i)$. One can consider more general policies (randomized or history-dependent). However, it has been shown that for a large class of finite-state, finite-action MDPs there exists a deterministic policy that is optimal. For this reason, and for tractability, we restrict `jMDP` to consider only deterministic policies. The total cost criterion in (3) is only useful for problems with costs that converge to zero fast enough. `jMDP` also supports a discounted cost criterion, for a given discount factor $0 < \alpha < 1$, where each term in (3) is multiplied by α^n; or a long-run average cost where the objective is the per-period average of (3).

Let $\mathcal{A}(i)$ be the set of *feasible actions* that can be taken from state i, and $\mathcal{S}(i, a)$ the set of reachable states from state i when action a is taken. Then $\mathcal{A}(i)$, $\mathcal{S}(i, a)$, $c(i, a)$ and $p_{ij}(a)$, together with the cost criterion (and α if necessary) completely characterize a DTMDP. Note that $c(i, a)$ only needs to be

defined for each feasible action $a \in \mathcal{A}(i)$ and $p_{ij}(a)$ is only defined for each feasible action $a \in \mathcal{A}(i)$ and each reachable state $j \in \mathcal{S}(i, a)$.

4.1.2. Finite Horizon MDP Problems (FHMDP). Now consider a discrete-time problem that only goes on for a finite number of periods $N < \infty$, called the *horizon* of the problem. In this case, we allow the parameters to vary over time. Specifically, each period n may have different feasible actions, reachable states, transition probabilities and costs. Therefore, the policies considered are also allowed to vary over time, so an action is selected for each state-period combination. We only consider the total cost criterion for FHMDP.

4.1.3. Continuous Time Infinite Horizon MDP Problems (CTMDP). These are a continuous time analog of DTMDPs, where instead of periods and transition probabilities the system dynamics are determined by time-homogeneous transition rates $\lambda_{ij}(a) = \lim_{h \to 0} P\{X(t+h) = j | X(t) = i, A(t) = a\}$. The Markov property implies that the sojourn time in state i under action a is exponentially distributed. In this case, costs can be a lump sum for each time an action is taken, or can be continuously incurred at a rate dependent on state and action. We consider the total, discounted and average cost criteria for CTMDP.

4.1.4. Problems Modeled with Events. All the previous problem types can be modeled in a more intuitive way using events. The `jMDP` module includes classes to model each of these problem types with or without events. We label the event-based version of each model by adding the suffix *ev*, e.g. FHMDPEv. Modeling MDPs with events implies that the transitions can be conditioned on certain events occurring at each period [Becker et al. 2004; Mahadevan et al. 1997; Feinberg 2004].

For example, consider a DTMDP, and let e be an event that triggers a state transition. Then the costs, reachable states, and transition probabilities will depend not just on the state and action taken, but also on the potential events e that can occur at each state. The value function (3) can also be written in terms of the event-dependent parameters. It has been shown [Mahadevan et al. 1997] that this DTMDP with events is equivalent to the standard DTMDP model presented earlier. `jMDP` also supports models where actions depend on the events, though we omit the details here in the interest of space.

4.2. Features and Algorithms

The `jMDP` module provides two functionalities to the user. First, it allows the user to model MDPs of all the types described in the previous section, in an efficient and intuitive manner. That is, following the same steps one would usually follow when defining the mathematical model. Second, it provides a suite of methods to solve such MDPs. In this section we discuss the algorithms and features associated with each of these functionalities.

4.2.1. Modeling Features and Algorithms. When modeling FHMDP or DTMDP the analyst generally follows these steps: characterize the states of the system, the events that can occur at each state, and the actions considered in each state; compute the conditional probability distribution of the destination state once an action has been taken and a given event occurs; and determine the immediate cost for every state, event, and action. `jMDP` allows the user to model MDPs by following these steps.

Our framework defines a computational class or method for each of these mathematical elements, which the user has to extend or implement in order to represent the model. To achieve this we leverage jMarkov's integrated structure and use the `Basic` package described in Section 2.3.2, which includes the abstract classes `Action`, `PropertiesAction`, and `Actions` to describe the actions in an MDP, analogous to the classes `State`, `PropertiesState`, and `States` to describe the states. We should note that this integrated structure means that if any new features, such as a class for novel state representations are added to the jMarkov `Basic` package, they become immediately available in `jMDP` and the rest of the modules.

The classes specific for MDP modeling are provided in a separate modeling package. Here we follow a hierarchical class structure where attributes and methods that are common to various classes are included only in the classes at the top of the hierarchy. This structure simplifies the extensibility of the module as new classes can exploit the methods already implemented in the existing classes. At the top of the hierarchy we have the abstract class `MDP`, which defines the basic procedures for MDP models, such as the `setSolver` and `getSolver` methods to define and access the solver, respectively. Then as we go down the hierarchy we implement more specific methods, for example the abstract class `InfiniteMDP` extends `MDP` and provides a number of methods, including for example a method to define feasible actions (`feasibleActions`). Finally, at the bottom of the hierarchy we have classes specific for each problem type described in Section 4.1, which exploit the methods provided by all the classes above it in the hierarchy. For example, to model a DTMDP, described in Section 4.1.1, the class `DTMDP`, has

ALGORITHM 2: MDPBuildRS algorithm

Input: $\mathcal{U} = \{i_0\}$, /* Input: the initial state */
Output: $\mathcal{S}$ /* Outputs: the state space */
$\mathcal{S} = \emptyset$;
while $\mathcal{U} \neq \emptyset$ **do**
  Choose an $i \in \mathcal{U}$;
  $\mathcal{S} := \mathcal{S} \cup \{i\}$;
  $\mathcal{U} := \mathcal{U} \backslash \{i\}$;
  foreach $a \in$ *feasibleActions*(i) **do**
    foreach $j \in$ *reachable*(i, a) **do**
      if $j \notin \mathcal{S} \cup \mathcal{U}$ **then** /* If j is unexplored add it to $\mathcal{U}$ */
        $\mathcal{U} := \mathcal{U} \cup \{j\}$;
      end
    end
  end
end

methods to define reachable states $\mathcal{S}(i, a)$ (`reachable`), immediate costs $c(i, a)$ (`immediateCost`), and transition probabilities $p_{ij}(a)$ (`prob`), but also uses methods from the superclass `InfiniteMDP`, such as `feasibleActions` to define the feasible actions $\mathcal{A}(i)$.

In order to build an DTMDP or FTMDP model in `jMDP` the user must extend the corresponding model class, as well as the `State` and `Action` classes to define the corresponding states and actions. Alternatively, users can exploit the existing `PropertiesState` or `PropertiesAction` classes, which define the state and action, respectively, as arrays of integers. Finally, the user has to implement the `feasibleActions`, `reachable`, `prob` and `immediateCost` methods.

With the definitions above, `jMDP` generates the corresponding MDP model. To achieve this, we extend the state-space exploration algorithm (*BuildRS*) described in Section 2.2.1, as depicted in Algorithm 2. Namely, instead of considering the possible events in each state, we incorporate the feasible actions and the reachable states defined in the MDP as the mechanisms to explore the state space. As we consider MDP models with a finite number of states and actions, it is possible to explore all the states in the MDP after a finite number of steps.

Modeling with events. If the user prefers to model a DTMDP using events, she additionally has to define a set of active events at each state i when action a is taken, $\mathcal{E}(i, a)$, and the probabilities of each event e happening at state i, $q(i, e)$. Similarly, the user must define the reachable states conditional on each event $\mathcal{S}(i, a, e)$, and define event dependent costs $c(i, a, e)$ and probabilities $p_{ij}(a, e)$. When the user asks `jMDP` to generate the model, the module automatically calculates the non-event-dependent versions of the model parameters as

$$c(i,a) = \sum_{e \in \mathcal{E}(i,a)} c(i,a,e) \cdot q(i,e), \quad p_{ij}(a,e) = \sum_{e \in \mathcal{E}(i,a)} p_{ij}(a,e) \cdot q(i,e), \quad \mathcal{S}(i,a) = \bigcup_{\mathcal{E}(i,a)} S(i,a,e).$$

Then `jMDP` executes Algorithm 2 to generate the state space, and the solver uses the non-event-dependent version of the parameters. In this case the user must extend the corresponding class with the `ev` suffix, e.g., `DTMDPEv`. A detailed example of modeling a DTMDP with events in `jMDP` is presented in Section 7. A similar process is followed when modeling FHMDPs with events.

Modeling with actions that depend on events. For models where actions are allowed to depend on events, the user follows the same process as above. However, in this case Algorithm 2 requires an additional modification to perform the required state expansion, because the state space of the underlying MC is made of all possible state-event combinations. To handle this `jMDP` uses the class `StateEvent` to represent state-event pairs. For each state in the initial set $\mathcal{U}$, `jMDP` generates a set of state-event pairs $\mathcal{U}'(i)$ and defines a new initial set $\mathcal{U}'$, respectively as

$$\mathcal{U}'(i) = \bigcup_{e \in \mathcal{E}(i)} \{i, e\}, \quad \text{and} \quad \mathcal{U}' = \bigcup_{i \in \mathcal{U}} \mathcal{U}'(i).$$

A similar modification is applied to the `reachable` method, so that for each state-event pair $\{i, e\}$ it returns all the reachable state-event pairs. Thus, the *MDPBuildRS* Algorithm receives $\mathcal{U}'$ as the initial set, and uses the modified version of the `reachable` method to do the state space expansion. Before solving the model, `jMDP` calculates $c(i, a)$ and $p_{ij}(a)$ for the new state space of state-event pairs. As a result, when the user asks for the solution, an action is assigned to each state-event pair instead

of just a state. This makes the state augmentation transparent to the user, as this process is completely handled by jMDP, and avoids distorting the modeling process with this technical feature. This is handled by the jMDP module by means of a family of classes with the suffix EvA, for example DTMDPEvA.

Continuous-time models. When modeling a CTMDP the user faces the additional challenge of having to transform the problem into an equivalent DTMDP, because the standard solution algorithms for MDPs cannot be applied to CTMDPs directly. Thus, the user typically needs to model the CTMDP, find the equivalent DTMDP, solve the equivalent DTMDP, and convert the obtained solution back to the original CTMDP. Instead, the jMDP module allows the user to model the problem directly as a CTMDP. jMDP automatically determines the equivalent DTMDP using the standard Uniformization algorithm [Serfozo 1979], solves the resulting DTMDP model using an appropriate DTMDP solver and presents the solution to the user in terms of the original CTMDP problem, making the entire uniformization process seamless. This algorithm is implemented in a special class called CT2DTConversion. The algorithm is available for continuous-time models with or without events. To model CTMDPs the user must extend the classes CTMDP, CTMDPEv or CTMDPEvA.

4.2.2. Solvers. The solvers package follows a similar hierarchical structure as the modeling package. At the top we have the abstract class called Solver, which defines methods such as solve that need to be implemented by any solver. Next, we include intermediate abstract classes for families of solvers, in order to define methods used often only once, making the most of the object-oriented architecture of jMarkov. For example the class AbstractDiscountedSolver defines a method called future, which calculates the expected one-step discounted value for a state-action combination. This method is used by all the solvers for discounted problems, but not by other solvers. In the following we detail the solution algorithms implemented in jMDP, each of them corresponding to a class at the bottom of the hierarchy in the Solvers package.

The backward induction algorithm [Bellman 1957] is the default algorithm for FHMDPs. The class FiniteSolver implements it to compute and return an optimal policy. The value iteration algorithm [Puterman 1994] is implemented in the class ValueIterationSolver. This is set as the default solver for infinite-horizon discounted problems, and allows activating or deactivating the Gauss-Seidel modification [Hastings 1971], and the use of error bounds [Bertsekas 1995]. For discounted problems, the policy iteration algorithm [Puterman 1994] has also been implemented in the class PolicyIterationSolver, where the MTJ library [Heimsund 2003] is used for the matrix operations. This class also allows the user to switch to the modified policy iteration algorithm [Puterman and Shin 1978] by simply setting a parameter and specifying the number of iterations for the internal loop of the algorithm.

For the average-cost optimization criterion, the relative value iteration algorithm [Puterman 1994] is set as the default. The algorithm is implemented in the class RelativeValueIterationSolver. For this type of problems we also provide an implementation of policy iteration and modified policy iteration in the class PolicyIterationAvgSolver. Finally, for the total-cost infinite-horizon problem, the default solver is the stochastic shortest path approach to MDPs [De Alfaro 1999], which we implement in the class StochasticShortestPathSolver.

Additionally, jMDP supports the use of linear programming (LP) software packages to solve infinite horizon problems under either the average cost or discounted cost criteria. In this case, one defines an equivalent LP model of the problem at hand, solves its dual, and constructs an optimal deterministic policy by observing which variables take strictly positive values. A detailed explanation of our implementation can be found in [Bello and Riaño 2006]. In particular, the classes MpsLpAverageSolver and MpsLpDiscountedSolver can be used for solving average and discounted cost problems, respectively. Each of these classes builds the dual LP model of the corresponding MDP in a Mathematical Programming System (MPS) file, which can in turn be read by an LP solver to solve the problem. To facilitate this process these classes are abstract and must be extended to communicate with the specific LP solver. We provide examples using the free QSopt LP solver [Applegate et al. 2003] and XpressMP [Dash Optimization 2005].

Finally, the extensible nature of the jMarkov framework allows the user to add new solver classes. This can include writing custom solvers for a specific class of problems, e.g. exploiting the problem structure; or creating a simple solver class to link to an existing solver the user has access to. Further details on the architecture of the module can be found in the jMDP documentation available at [jMarkov website 2016].

4.3. Related Work

To the best of our knowledge, all the tools currently available for MDPs are only *solvers*. jMDP is unique because it offers *modeling* capabilities as well as *solvers* for MDPs. More precisely, the solvers available

in existing tools require the user to provide a three-dimensional array of transition probabilities and a matrix of costs as input, together with the horizon and cost criterion. Due to the large number of state-action combinations, or the even larger number of state-event-action combinations when the problem is modeled with events, the process of constructing these inputs is generally cumbersome and error prone. In order to use existing tools, the user must perform this process separately on her own. However, with `jMDP` this step is unnecessary as the user only defines the states, events, and actions, along with the rules that determine which events and actions are available at each state, and the corresponding transition probabilities and costs. `jMDP` takes care of building the state space and all other parameters of the model in the background. Furthermore, available tools do not support models with events, nor actions that depend on events. Thus, if a user builds a mathematical model using events, she must convert that model into a model without events and convert the conditional probabilities and costs into unconditional parameters in order to use the available tools. Similarly, if she wants to allow actions to depend on events, she must build the new state space of state-event pairs on her own, and convert the parameters accordingly. With `jMDP` these conversions are carried out in the background, so the process is seamless for the user.

Another important distinction between `jMDP` and existing solvers is its implementation of the Uniformization algorithm, which allows users to model and solve CTMDP problems. All the algorithms commonly used to solve CTMDP problems require finding the DTMDP analogue of the original problem, solving that problem, and then converting the solution back to the terms of the original CTMDP. In order to use any of the currently available tools to solve MDPs for a continuous-time problem, the user needs to find the DTMDP analogue *a priori* and then use the solver on that problem. With `jMDP` this process is seamless, to the point that the user does not need to manipulate the discrete-time problem at all, since the solution is presented in terms of the original CTMDP.

Currently available general purpose MDP solvers include MDPtoolbox [Chadès et al. 2014], available for MATLAB, R, GNU Octave, and Scilab, which offers multiple algorithms for solving DTMDP. PyMDPtoolbox [Cordwell 2013] is a Python implementation of that toolbox. Another tool for MATLAB, called MDP Toolbox [Murphy 2002] is similar to MDPtoolbox, but has fewer algorithms implemented and is no longer supported. These three solvers require the state and action spaces to be a subset of the integers, which can make parsing a solution to the original state space a hassle if such representation is not natural for a specific problem. This is not an issue in `jMDP`, where the user can define states and actions more generally. These tools also lack `jMDP`'s object-oriented architecture, which allows the user to easily modify the model, which is useful, for example, to do sensitivity analyses or to calculate additional measures of performance. Furthermore, thanks to `jMDP`'s extensibility a user could implement a new solver that calls any of the functions in these toolboxes to solve models constructed using `jMDP`.

Besides the aforementioned general purpose toolboxes, there exist some tools that implement a specific algorithm or focus on a specific problem. SPUDD [Hoey et al. 1999] uses the SPUDD algorithm, which only works for discounted DTMDPs and requires the state to be represented as a boolean vector, limiting its applicability. PRISM [Kwiatkowska et al. 2011] is a tool for probabilistic model-checking, which can check several structural properties of MDPs, DTMCs and CTMCs, but is not designed to determine optimal policies or performance measures, thus it is outside the scope of `jMDP`. Caylus [Teichteil-Königsbuch et al. 2010] implements several heuristic algorithms for planning problems, using aggregation-disaggregation techniques to get solutions quickly. However, its scope is limited to problems that conform to that particular structure, and it offers no guarantee of optimality. We should note here that there are several tools available for solving Partially Observable Markov Decision Processes, which in some cases can be used to solve MDPs. However, since these usually implement approximation algorithms and offer limited support for pure MDP problems, we consider them to be outside the scope of the `jMDP` module.

5. THE JPHASE MODULE

Phase-type (PH) distributions [Neuts 1981; Latouche and Ramaswami 1999] are a powerful tool for stochastic modeling as they are able to represent a broad range of probabilistic behaviors. Further, PH distributions can be introduced into Markovian models, which are suitable for efficient numerical analysis, replacing the more common exponential distribution. The jMarkov framework includes the `jPhase` module to represent PH distributions, as well as to support their inclusion in CTMC models. In addition, `jPhase` provides methods to obtain PH distributions from data traces and to generate PH random variates. The next sections introduce PH distributions and detail the features of `jPhase`.

5.1. Background

A continuous PH distribution [Neuts 1981] is defined as the time until absorption in an absorbing CTMC with m transient states. Letting $\boldsymbol{A}$ be the $m \times m$ subgenerator matrix associated with the transient states, and $\boldsymbol{\alpha}$ their initial probability distribution, the cumulative distribution function of the time to absorption is given by $F(t) = 1 - \boldsymbol{\alpha} e^{\boldsymbol{A}t}\mathbf{1},\ t \geq 0$, and its moments can be obtained as $E[X^k] = k!\boldsymbol{\alpha}(-\boldsymbol{A}^{-1})^k\mathbf{1},\quad k \geq 1$. We refer to this distribution or the associated random variable as $\mathrm{PH}(\boldsymbol{\alpha}, \boldsymbol{A})$. Due to their Markovian description, PH distributions can be used instead of the exponential distribution in Markovian models, capturing more general behaviors [Latouche and Ramaswami 1999].

Analogously, discrete PH distributions represent the distribution of the number of steps until absorption in an absorbing Discrete Time Markov Chain (DTMC). Also, PH distributions, both continuous and discrete, enjoy being closed under a number of operations, including convolution, order statistics, and convex mixtures, among others. These closure properties can be exploited in the modeling of, for instance, manufacturing systems [Riaño 2002].

5.2. Features and Algorithms

The `jPhase` module offers three main sets of features: manipulation of PH distributions, including closure properties; fitting procedures to obtain PH distributions from data traces; and PH random-variate generation. These sets of features correspond to the three main packages in the module: `jPhase`, `jPhaseFit`, and `jPhaseGenerator`.

5.2.1. Manipulation and Closure. The central functionality of the `jPhase` module is the manipulation of PH variables. This entails the definition of PH variables as Java objects, and the computation of relevant metrics. `jPhase` thus offers methods to compute moments, such as mean or variance, as well as to evaluate the cumulative distribution function or the density probability function of a PH random variable. As the set of PH distributions includes many common special cases, such as the exponential, Erlang or hyper-exponential distributions, `jPhase` also provides methods to define these special cases based on their specific parameters. Furthermore, `jPhase` offers dense and sparse storage for the parameters of the PH distribution, vector and matrix, making use of the classes available in the MTJ library [Heimsund 2005].

Since PH variables have a number of closure properties that can be used for modeling purposes, `jPhase` also provides support for a broad set of these properties. For instance, the minimum or the maximum of a set of PH variables is also PH-distributed, as is the sum of two PH variables. To illustrate this, let us consider a production line where the service is made up of two stages. The first stage follows a $\mathrm{PH}(\boldsymbol{\alpha}, \boldsymbol{A})$ distribution with parameters

$$\boldsymbol{\alpha} = [0.5\ \ 0.3\ \ 0.2] \text{ and } \boldsymbol{A} = \begin{bmatrix} -4 & 2 & 1 \\ 1 & -3 & 1 \\ 2 & 1 & -5 \end{bmatrix},$$

while the second stage is an $\mathrm{Erlang}(n, \lambda)$ distribution with parameters $\lambda = 1.5$ and $n = 2$, and the arrivals follow a Poisson process. Assuming the production line faces a traffic intensity (the ratio between the arrival and the service rates) of 0.5, we are interested in knowing the probability that the incoming jobs face a waiting time between 1 and 2 time units. The following code snippet shows how this can be obtained with `jPhase`.

```
1  DenseMatrix A = new DenseMatrix(
2          new double[][] { {-4,2,1} , {1,-3,1} , {2, 1,-5} } );
3
4  DenseVector alpha = new DenseVector(
5          new double[] {0.5, 0.3, 0.2});
6
7  DenseContPhaseVar v1 = new DenseContPhaseVar(alpha, A);
8
9  ContPhaseVar v2 = DenseContPhaseVar.Erlang(1.5, 2);
10
11 ContPhaseVar v3 = v1.sum(v2);
12
13 double rho = 0.5;
14
15 ContPhaseVar v4 = v3.waitingQ(rho);
16
17 double prob = v4.prob(1,2);
```

Here the first two lines define the parameters $\boldsymbol{\alpha}$ and $\boldsymbol{A}$, while the third line builds a dense (storage) representation of this PH variable. The fourth line defines the Erlang distribution for the second stage of service. Next we build the full representation of the service time (`v3`) by summing the variables corresponding to the two service stages just defined. We can then apply another closure property since

the waiting time distribution in a first-come-first-serve M/PH/1 queue (Poisson arrivals, PH services, 1 server) is also PH. We thus call the `waitingQ` method in `jPhase` to obtain the PH representation of the waiting time as variable `v4`. Finally, we use the method `prob` to obtain the desired probability.

5.2.2. Fitting. To incorporate a PH distribution in an MC model a necessary step is to determine the parameters $(\boldsymbol{\alpha}, \boldsymbol{A})$. Consider for instance the parameterization of a queueing model representing a server in a software application. The simplest option to model the processing times at this server is to assume they follow an exponential distribution, and set the parameter (rate) of the exponential distribution to match the measured mean processing time. However, the exponential assumption may be far from the actual processing times, for instance the real distribution may have a much shorter or longer tail than the one assumed by the exponential distribution. To incorporate this more general behavior we can instead assume PH-distributed processing times, for which we must set the parameters $(\boldsymbol{\alpha}, \boldsymbol{A})$. `jPhase` therefore offers a set of *fitting* routines, which aim to find the parameters $(\boldsymbol{\alpha}, \boldsymbol{A})$ that best fit a set of measurements or statistical data. These routines are provided in the `jPhaseFit` package, which also provides a set of interfaces, abstract classes, and other methods to allow the interested user to incorporate new fitting methods. We now describe the fitting methods already provided by the `jPhaseFit` package, which we classify in maximum-likelihood and moment-matching techniques.

Maximum-Likelihood Algorithms. Maximum-likelihood methods employ optimization methods to find the parameters $(\boldsymbol{\alpha}, \boldsymbol{A})$ of a PH distribution that are most-likely to generate a given data trace. All the methods of this type implemented in `jPhase` rely on the Expectation-Maximization (EM) algorithm [Dempster et al. 1977]. `jPhaseFit` implements, in class `EMPhaseFit`, the method proposed in [Asmussen et al. 1996], which considers the general class of continuous PH distributions as the target for the fitting method. This method is however computationally expensive as the E-step of the EM algorithm requires solving a set of $n(n+2)$ linear differential equations for a distribution of n phases. To deal with this issue, alternative methods consider specific subclasses of PH distributions as their target set. The method in [El Abdouni Khayari et al. 2003], implemented in class `EMHyperExpoFit`, focuses on hyper-exponential distributions, which are convex mixtures of exponential distributions and can be represented by a PH distribution with a diagonal matrix $\boldsymbol{A}$. A similar, though more general, approach was proposed in [Thümmler et al. 2005], which targets hyper-Erlang distributions. These are convex mixtures of Erlang distributions that include the hyper-exponential as a special case, but also allow for hypo-exponential behaviors, such as the Erlang distribution. This method is implemented by the `EMHyperErlangFit` class.

Moment-Matching Algorithms. Moment-matching methods aim to find the parameters $(\boldsymbol{\alpha}, \boldsymbol{A})$ such that the PH distribution has a set of predefined moments (mean, variance, asymmetry, etc). Such methods are very useful in analyzing the impact that a given moment, for instance the variance, of a certain quantity has on the measures of performance. In the production line example, one can be interested in determining the impact that the variability of the processing times, measured by their variance, has on the total time to produce an item, including queuing times. `jPhaseFit` offers four methods of this type, three for continuous PH distributions, and one for discrete ones. The method in [Telek and Heindl 2002] matches the first three non-centered moments with a two-phase PH distribution, as long as these are representable. `jPhaseFit` implements both the continuous and discrete variants of this method in the classes `MomentsACPH2Fit` and `MomentsADPH2Fit`. `jPhaseFit` also implements, in class `MomentsECCompleteFit`, the method proposed in [Osogami and Harchol 2006], which determines the minimum number of phases needed to represent any set of three non-centered moments attainable by a distribution with non-negative support. This feature is shared by the method proposed in [Bobbio et al. 2005], which is implemented in class `MomentsACPHFit`.

5.2.3. Random-Variate Generation. While the main focus of jMarkov is on the numerical solution of MC models, we also recognize that an alternative tool for modeling and analysis is to use stochastic simulation. In fact, in many applications simulation is more appropriate because of the complex relationships between different stochastic variables. While PH distributions have gained popularity, they are still not mainstream and are therefore not supported by most simulation tools. `jPhase` therefore includes support for PH random-variate generation in the `jPhaseGenerator` package. The classes in this package offer implementations of the methods proposed in [Neuts and Pagano 1981] for both continuous and discrete PH distributions. Since jMarkov is distributed as a Java library, its random-variate generators can be incorporated in simulation models developed in this language, for instance using the SSJ library [L'Ecuyer and Buist 2005].

5.3. Related Work

All tools available for PH distributions concentrate on *fitting*, and most are standalone implementations accompanying the papers where the corresponding fitting method was introduced. For instance, EMPht [Olsson 1998] implements the method in [Asmussen et al. 1996] to fit general continuous PH distributions, while PHFit implements the fitting methods in [Horváth and Telek 2002] for both discrete and continuous PH distributions. Also, G-Fit implements the method proposed in [Thummler et al. 2006] for hyper-Erlang distributions, a sub-class of PH distributions, and HyperStar provides an implementation of the methods in [Reinecke et al. 2012] with a graphical interface to fine-tune the results. Different from the tools above, `jPhase` provides a range of methods for PH fitting, which allows the user to select the one that best fits his needs, or to implement new methods.

A more comprehensive tool is Butools [Butools 2016], which also offers a range of methods for fitting PH distributions, as well as methods to compute moments and probability functions. As mentioned in Section 3.3, Butools also offers solvers for QBD processes and other structured MC models. However, `jPhase` is unique in implementing a wide set of closure properties useful for modeling, as well as in providing random-variate generation methods. More importantly, `jPhase` is part of the jMarkov modeling framework, which allows the user to exploit PH variables, built from data and/or by means of closure properties, to construct finite MC and QBD models from first principles. The next section provides an illustrative example.

6. MODELING PRIORITY QUEUES WITH JMARKOV

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 [jMarkov website 2016], but to highlight some of the key steps in modeling with jMarkov. We also report the execution times obtained with jMarkov, and compare them to an alternative tool.

6.1. A priority-queue model

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 $(\boldsymbol{\alpha}^{(i)}, \boldsymbol{A}^{(i)})$. 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 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;
```

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 GeomProcess<PriorityQueueMPHPHPreemptState, PriorityQueueMPHPHPreemptEvent>{
2          double lambda_hi;
3          double lambda_low;
4          PhaseVar servTime_hi;
5          PhaseVar servTime_low;
6          int bufferCapacity;
7  }
```

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 $-\boldsymbol{A}^{(1)}\mathbf{1}$ 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, servTime_hi, servTime_low,
14                                                                                                          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 $\boldsymbol{A}_0$, $\boldsymbol{A}_1$, or $\boldsymbol{A}_2$ in (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 (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.

6.2. Numerical experiments

To validate the algorithms implemented in jMarkov we have performed exhaustive tests on jMarkov. In [jMarkov website 2016] we currently provide over 20 tests, in the form of jUnit [jUnit 2016] test cases, where we compare the results of many models against results obtained with alternative tools. These test cases cover the main module, as well as the `jQBD`, `jMDP`, and `jPhase` modules.

To illustrate the execution times that can be expected with jMarkov we make use of the priority queue example introduced earlier in this section. We set the arrival rates to achieve three levels of server utilization ρ, namely 0.1, 0.5, and 0.9. The utilization is the fraction of time that the server is busy and influences the execution time of the logarithmic-reduction algorithm used to solve the QBD model. Also, we set the service time distributions to have 2 phases, and we consider values for the buffer capacity C between 100 and 2000. Table II reports the size m of the $\boldsymbol{A}_i$ matrices that define the QBD in (1). Table II also compares the execution times, in seconds, obtained with jMarkov against those obtained with SMCSolver [Bini et al. 2009], which executes in MATLAB. These times were obtained on a MAC with a 2.9 GHz Intel Core i7 with 2 cores, 8 GB of memory, and running OS X 10.11.4. First we note that jMarkov is able to solve large systems, with block sizes up to 4000. jMarkov is however about 2-5 times slower than SMCSolver. This comes at no surprise since SMCSolver relies on the very efficient matrix manipulation in MATLAB. In spite of this, jMarkov is well-suited to consider middle-sized problems, requiring less than a second to solve problems with block size 200, and just 30 seconds for problems with block size 1000.

Table II. Execution times (sec) with jMarkov and SMC Solver for the priority queue model

C	m	jMarkov ρ 0.1	jMarkov ρ 0.5	jMarkov ρ 0.9	SMCSolver ρ 0.1	SMCSolver ρ 0.5	SMCSolver ρ 0.9
100	202	0.38	0.36	0.34	0.06	0.07	0.08
200	402	1.76	2.26	2.49	0.44	0.37	0.45
500	1002	31.12	31.10	29.87	13.99	10.83	9.82
1000	2002	252.5	292.4	297.7	63.70	144.9	134.6
2000	4002	2151	2298	2586	435.5	659.3	997.7

As we have highlighted, the key benefits of jMarkov lie in its modeling capabilities, but we have seen above that its solvers are also well-suited for middle-sized problems. In addition, the flexibility of jMarkov, and its implementation in Java, offer a number of options to solve larger problems, or to exploit the reduced execution times offered by tools like SMCSolver on MATLAB. It is possible for instance to generate the MC/QBD model in jMarkov, save the associated matrices in files, which are then loaded in a different tool to solve the model. Alternatively, in the following code snippet we show how to directly import the jMarkov routines from MATLAB to exploit the solvers in SMCSolver. The first step is to include the jMarkov library in the static Java classpath of MATLAB. This allows us, as shown in the first 4 lines of the code snippet, to import the jMarkov routines, as well as the MTJ library for some matrix manipulations. In particular, the third line imports the model `PriorityQueueMPHPHPreempt` we defined in the previous section. Lines 6-14 define the model parameters in a similar manner as in the previous section, exploiting the `jPhase` routines to fit PH distributions. Line 16 creates the model object and line 17 generates the model parameters. The next step, in lines 19-21, is to extract the matrices that define the QBD from the jMarkov `model` object. Finally, we use these matrices to call the routines in SMCSolver to solve the model, as in lines 23-24. This illustrates that the modeling capabilities of jMarkov can be exploited to build complex models, and the user can easily employ alternative solvers if this is required or preferred.

```
1  import jphase.fit.EMHyperErlangFit;
2  import jphase.fit.MomentsACPHFit;
3  import examples.jmarkov.PriorityQueueMPHPHPreempt;
4  import no.uib.cipr.matrix.Matrices;
5
6  lambda_hi = 0.2; lambda_low = 0.2;
7  bufferCapacity = 10;
8
9  data = csvread('data/W2.txt');
10 fitter_hi = EMHyperErlangFit(data);
11 servTime_hi = fitter_hi.fit(4);
12
13 fitter_low = MomentsACPHFit(2, 6, 25);
14 servTime_low = fitter_low.fit();
15
16 model = PriorityQueueMPHPHPreempt(lambda_hi, lambda_low, servTime_hi, servTime_low, bufferCapacity);
17 model.generate();
18
19 As = model.getAMatrices(); Bs = model.getBMatrices();
20 A0 = Matrices.getArray(As(1)); A1 = Matrices.getArray(As(2)); A2 = Matrices.getArray(As(3));
21 B00 = Matrices.getArray(Bs(1)); B01 = Matrices.getArray(Bs(2)); B10 = Matrices.getArray(Bs(3));
22
23 [G,R] = QBD_LR(A2,A1,A0);
24 pi = QBD_pi(B10,B00, R, 'Boundary', [B01; A1 + R*A2]);
```

7. MODELING AND SOLVING AN INVENTORY PROBLEM WITH JMARKOV

In this section we present an inventory management example to illustrate the process of modeling and solving an MDP with the `jMDP` module of jMarkov. We also show how a user can implement an MDP model using `jMDP` and then, due to jMarkov's flexibility, call this implementation from a different software program and use a different tool to solve it.

7.1. An inventory management model

Consider the following problem. A car dealership sells only one type of car and uses a weekly (periodic) inventory review system. Each car is bought at a cost c and sold at a price p. The dealership must pay a fee K per truck for carrying the cars from the distributor to its location, and each truck can carry at most L cars. The dealership has a maximum capacity of M cars, and orders arrive instantly. If a customer places an order and there are no cars available, the sale is lost. We assume a fixed inventory holding cost of h per car and week. The demands for cars each week, D_n, are independent, identically distributed Poisson random variables with a mean of λ cars per week. The objective is to find an optimal ordering policy that maximizes weekly profits.

The problem is an infinite-horizon, discrete-time stochastic decision-making problem, and the objective is to minimize the long run average cost. The time periods are weeks because the inventory review occurs weekly. We model it as DTMDP with events, as this description is more natural than without events (jMDP supports both options). In the following we describe the step-by-step mathematical modeling process and the implementation in jMDP.

Define the states. Let X_n be the level of physical inventory at the end of week n. The state space is $\mathcal{S} = \{0, 1, ..., M\}$. In the code below we declare the class InvLevel, which represents the state. It extends PropertiesState which is used to represent states as arrays of integers (in this case the array has only one entry). In line 2 we provide a constructor for the class. In the interest of space, we will only include key portions of the code. Ellipses (...) indicate that further code is used; in this case for example, the class includes methods such as getLevel to return the inventory level.

```
1 public class InvLevel extends PropertiesState {
2         public InvLevel(int k) {super(new int[] {k});}
3 (...) }
```

Define the potential actions. Let a_n represent the size of the order placed at the start of week n. In the code below we create the class Order, which represents the actions and extends Action. We define the field size in line 2 to represent the amount ordered, and in line 3 we provide an appropriate constructor.

```
1 public class Order extends Action {
2         private int size;
3         Order(int k) {size = k;}
4 (...) }
```

Define the events. Here the events are the random demands e_n that occur each week. Notice that events occur after action a_n is taken. The event definition below includes two variables. An integer d, represents the size of the demand. And a boolean variable greaterThan, which takes the value "true" if the demand d is greater than or equal to the total inventory $X_{n-1} + a_n$ at the beginning of the period, and "false" otherwise. Here we extend the class PropertiesEvent, which represents events as arrays of integers. In lines 3-5 we provide an appropriate constructor.

```
1 public class DemandEvent extends PropertiesEvent {
2         private boolean greaterThan;
3         public DemandEvent(int d, boolean greater) {
4                 super(new int[] { d });
5                 greaterThan = greater;}
6 (...)}
```

Define the MDP. This is a DTMDP, therefore we extend the class DTMDPev. When extending this class, we use the corresponding classes that represent the states, actions and events. In our example, InvLevel, Order and DemandEvent represent the states, actions, and events, respectively. In the following code CarDealerProblem is the class representing the problem, and it includes fields, not shown for brevity, for each problem parameter, namely maxInventory, truckSize, fixedCost, lambda, price, cost, holdCost, and truckCost.

```
1 public class CarDealerProblem extends DTMDPEv<InvLevel, Order, DemandEvent> {(...)}
```

Define the feasible actions. For each state i the feasible actions are those that do not exceed the dealership's capacity. The maximum feasible order in state i is thus $M - i$, an amount that we calculate in line 2 in the code below, and the feasible set of actions is $\mathcal{A}(i) = \{0, \ldots, M - i\}$. The method feasibleActions receives the state i as a parameter. In line 3 we create an empty set of actions. In the loop in lines 4-6 we add each of these actions to the set.

```
1 public Actions<Order> feasibleActions(InvLevel i){
2         int max = maxInventory - i.getLevel();
3         ActionsSet<Order> actionSet = new ActionsSet<Order>();
4         for (int n = 0; n <= max; n++){
5                 actionSet.add(new Order(n));
6         }
7         return actionSet;}
```

Define the active events. For each state i, and given that action a is taken, we have to define the events that can occur. For example, the zero-demand event is active in every state, while a demand equal to $i + a$ is indistinguishable from larger demands as it empties the available inventory. We define the method activeEvents, which starts by creating an empty set of events eventSet in line 2, to which we add the active events. Since each event is defined by both the amount demanded and the boolean greaterThan indicating whether the demand empties the inventory or not, we specify each event with these two parameters. In line 3 we add the event where the demand is at least equal to $i + a$ and

greaterThan is true. The loop in lines 4-6 adds events where the demands is less than $i + a$, and set greaterThan to false as the inventory remains positive after the demand event.

```
1 public Events<DemandEvent> activeEvents(InvLevel i, Order a) {
2         EventsSet<DemandEvent> eventSet = new EventsSet<DemandEvent>();
3         eventSet.add(new DemandEvent(i.getLevel() + a.getSize(), true));
4         for (int n = 0; n < i.getLevel() + a.getSize(); n++) {
5                 eventSet.add(new DemandEvent(n, false));
6         }
7   return eventSet;}
```

Define the set of reachable states Here we define the states that the MDP can transition to from state i, given that action a is taken and event e occurs. In the method reachable, displayed below, we create a new set of states in line 2. If the demand event is greater than $(i + a)$ the new state is 0, as depicted in line 4. Otherwise, if the demand is d, the only reachable state is $(i + a - d)$, as set in line 6.

```
1 public States<InvLevel> reachable(InvLevel i, Order a, DemandEvent e) {
2         StatesSet<InvLevel> stSet = new StatesSet<InvLevel>();
3         if (e.getGreaterThan())
4                 stSet.add(new InvLevel(0));
5         else
6                 stSet.add(new InvLevel(i.getLevel() + a.getSize() - e.getDemand()));
7         return stSet;}
```

Define the event probabilities. We condition on the event $d < i+a$. The probability of going from state i to a reachable state j when action a is taken is given by

$$p_{ij}(a) = \begin{cases} P\{D_n = d\} & \text{if } j = i + a - d,\ d < i + a, \\ P\{D_n \geq i + a\} & \text{if } j = 0,\ d \geq i + a, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

In the code below, demCCDF denotes the cumulative distribution function of the demand, and demPMF its probability mass function. These values have been previously generated and correspond to a Poisson distribution. The condition in line 2 is equivalent to the second case in (4), while the complementary case in line 4 is equivalent to the first case in (4).

```
1 public double prob(InvLevel i, DemandEvent e) {
2         if (e.getGreaterThan())
3                 return demCCDF[e.getDemand()];
4         return demPMF[e.getDemand()];}
```

Define the immediate cost As jMDP assumes a minimization objective function, we minimize the negative of the net profit, defined in the method immediateCost in the code below. The profit has three major components. First, the revenues, which are calculated as the selling price times the expected sales $p \times (\mathbb{E}[D_n] - L_{D_n}[i+a])$, where L_{D_n} is the first-order loss function of the demand distribution [Zipkin 2000]. The expected sales are calculated in lines 2 and 3 of the code below. Second, the ordering cost includes a charge per truck and a charge per car, and when the truck is only partially occupied the whole truck is charged, thus it is given by $K \left\lceil \frac{a}{L} \right\rceil + ca$. This cost is computed in lines 7 and 8 below. Third, the holding cost, which depends only on the state and is charged only when the stock is positive. Hence, it can be calculated as $h \times i$. The immediate cost is therefore given by:

$$c(i, a) = -\left(p \times (\mathbb{E}[D_n] - L_{D_n}[i + a]) - K \left\lceil \frac{a}{L} \right\rceil - c \times a - h \times i\right)$$

as calculated in lines 4 and 5 below.

```
1 public double immediateCost(InvLevel i, Order a) {
2         int maxSale = i.getLevel() + a.getSize();
3         double expectedSales = expDemand - demandLoss1[maxSale];
4         double netProfit = price * expectedSales - orderCost(a.getSize())- holdCost*i.getLevel();
5         return -netProfit;
6 }
7 double orderCost(int x) {
8         return truckCost * Math.ceil((double) x / truckSize) + x * cost;}
```

Generate and solve the model. In this method we set the values of the parameters to define a specific instance of the problem. As an example, the values of the parameters for this instance are $M = 10$, $L = 4$, $\lambda = 9$, $p = 1100$, $c = 500$, $h = 50$, $K = 1000$, which are set in lines 2 and 3 below. In the next lines we generate an instance of the problem with the given parameters and solve it. This is where the state-space search algorithm is executed to build the whole state space $\mathcal{S}$. Finally, we call the solver, and print the solution. We use the default Relative Value Iteration Algorithm, the solver takes the model object as input.

```
1 public static void main(String a[]) throws SolverException {
2         int maxInventory = 10; int truckSize = 4; double lambda = 9; double price = 1100; double cost = 500;
3         double holdCost = 50; int truckCost = 1000;
4         CarDealerProblem prob = new CarDealerProblem(maxInventory, truckSize, fixedCost, lambda, price, cost, holdCost, truckCost);
5         prob.solve();
6         prob.printSolution();
7 }
```

The results for this problem are stored as a `Solution` object, and the last line above prints the following optimal policy.

```
1 STATE       ------> ACTION
2 LEVEL 0     ------> ORDER 8 UNITS
3 LEVEL 1     ------> ORDER 8 UNITS
4 LEVEL 2     ------> ORDER 8 UNITS
5 LEVEL 3     ------> ORDER 7 UNITS
6 LEVEL 4     ------> ORDER 4 UNITS
7 (...)
```

This policy contains the optimal action to be taken in each possible state. For instance, if the current inventory level is 4 then it is optimal to order 4 cars. This example can be found in [jMarkov website 2016]. In fact, a finite horizon variation of this example is included as a one of the `jUnit` tests used to test the framework for correctness.

7.2. Modeling with jMarkov and solving with another tool

The flexibility of jMarkov, coupled with the fact that it is implemented in Java, provides the user with multiple alternatives for solving larger problems. The user can choose to model and solve the problem using only `jMDP` as in the previous section. Alternatively, the user can build the model with `jMDP`, exploiting the modeling capabilities of the module, and then use a different tool for the solution step. This option can be carried out in three ways: (i) by generating the model in `jMDP`, exporting the parameters and then importing them to another tool; or, (ii) by writing a solver class in Java using the jMarkov framework, which invokes the desired solver tool (this is the way LP solvers work in `jMDP`); or, (iii) by importing a model constructed with `jMDP` into another tool in order to solve it. In Section 6.2 we followed the third option to use SMCSolver in MATLAB to solve a QBD model built with jMarkov. Here we follow a similar procedure to import the `jMDP` model developed in the previous section into MATLAB and use MDPtoolbox [Chadès et al. 2014] to solve it.

The code below shows how to import a `jMDP` model into MATLAB and use the functions provided by MDPtoolbox to find a solution. Line 1 imports the model class `CarDealerProblem`, which we described in detail in Section 7.1. Lines 3 and 4 define the model parameters, and Line 6 creates the model object. Lines 7 and 8 generate the model parameters that the MDPtoolbox solver uses as input. Specifically, the method `getTheP` generates a 3-dimensional array of transition probabilities $p_{ij}(a)$, whereas the method `getTheR` generates a matrix of immediate costs $c(i, a)$. Notice that the cost matrix is multiplied by -1 because the default setting of MDPtoolbox is maximization, while the costs are calculated for a minimization problem. Finally, line 10 calls one of the MDPtoolbox solver functions to solve the problem and return the optimal policy, long-run average cost and solution time.

```
1  import examples.jmdp.CarDealerProblem;
2
3  maxInventory = 10; truckSize = 4; lambda = 7.0; truckCost = 800.0;
4  price = 1100.0; cost = 500.0; holdCost = 50.0;
5
6  model=CarDealerProblem(maxInventory, truckSize, truckCost, price, cost, holdCost, lambda);
7  P=model.getTheP();
8  R=-1*model.getTheR();
9
10 [policy, cost, cpu_time] = mdp_relative_value_iteration (P, R);
```

This example illustrates how the modeling capabilities of jMarkov can be exploited to build a complex model using events and to solve it with a tool that does not support MDP models with events. But, because `jMDP` automatically converts the event-dependent model into a DTMDP without events and automatically calculates the non-event-dependent version of the parameters, the process is completely seamless for the user. This could not have been achieved without jMarkov. If the user only had MDPtoolbox available, she would have had to manually generate the parameters for the MDP with events and transform it into a DTMDP without events in order to solve it.

8. APPLICATIONS AND EXTENSIONS

To conclude the paper we would like to mention some applications of the jMarkov framework to specific problems, which exploit the modeling capabilities of jMarkov to build complex MC, QBD and MDP

models from basic rules. We also highlight some extensions that rely on the ease-of-extensibility built in to jMarkov. [Bello and Riaño 2006] discusses how jMarkov can be effectively connected to commercial software packages to solve MDPs as Linear Programming problems. [Medina et al. 2007] uses jMDP to build a model that reduces the cost of issuing a mortgage-backed security by changing the structure of the security issued. The model was implemented in a real-world instance, using the original data of a Colombian securitizing firm, which allowed the authors to determine the different levels of improvement attained by the model. [Riaño et al. 2006] uses jMarkov to model a bucket-brigades system [Bartholdi and Eisenstein 1996] with stochastic processing times and to compare it to the basic deterministic model in the original paper. [Silva et al. 2009] uses jMarkov to model congestion in manual order-picking-systems and compare layouts with narrow and wide aisles. [Ucrós Ospino 2012] uses jMarkov to consider PH distributions in the modeling of a semiconductor manufacturing system as a queueing network. Most recently, [Gomez et al. 2013] uses jPhase to fit PH distributions to travel and service time data, as part of a stochastic vehicle routing problem. These results highlight the ability of jMarkov to support researchers in the development and solution of complex models.

References

Advanced Logistic Department 2013. *RAM Commander, Version 8.3*. Advanced Logistic Department. http://aldservice.com/en/reliability-products/markov.html

D. Applegate, W. Cook, S. Dash, and M. Mevenkamp. 2003. *QSopt Reference Manual*. http://www2.isye.gatech.edu/~wcook/qsopt/index.html

S. Asmussen, O. Nerman, and M. Olsson. 1996. Fitting Phase Type distributions via the EM algorithm. *Scandinavian Journal of Statistics* 23 (1996), 419,441.

J. J. Bartholdi and D. D. Eisenstein. 1996. A production Line that Balances itself. *Oper. Res.* 4, 2 (1996), 21–34.

R. Becker, S Zilberstein, and V. Lesser. 2004. Decentralized Markov Decision Processes with Event-Driven Interactions, In Autonomous Agents & multi Agent Systems. *AAMAS* (July 2004).

R. Bellman. 1957. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey.

D. Bello and G. Riaño. 2006. Linear Programming solvers for Markov Decision Processes.. In *Proceedings of the 2006 IEEE Systems and Information Engineering Design Symposium*, Michael DeVore (Ed.). 93–98. http://www.sys.virginia.edu/sieds06/papers.html

D. Bertsekas. 1995. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Massachusetts.

D. Bini, G. Latouche, and B. Meini. 2005. *Numerical Methods for Structured Markov Chains*. Oxford.

D. Bini, B. Meini, S. Steffé, and B. Van Houdt. 2009. Structured Markov chains solver: tool extension. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 20.

A. Bobbio, A. Horvath, and M. Telek. 2005. Matching three moments with minimal acyclic Phase Type distributions. *Stochastic Models* 21 (2005), 303–326.

Butools. 2016. http://webspn.hit.bme.hu/~telek/tools/butools/index.php. (2016).

I. Chadès, G. Chapron, M.J. Cros, F. Garcia, and R. Sabbadin. 2014. MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems. *Ecography* 37, 9 (2014), 916–920.

G. Ciardo. 2000. Tools for formulating Markov models. In *Computational Probability*, Winfried K. Grassman (Ed.). Kluwer's International Series in Operations Research and Management Science, Massachusetts, USA.

G. Ciardo, III Jones, R.L., R. M. Marmorstein, A.S. Miner, and R. Siminiceanu. 2002. SMART: stochastic model-checking analyzer for reliability and timing. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. 545–.

S. Cordwell. 2013. PyMDPtoolbox. Available at http://code.google.com/p/pymdptoolbox/. (2013).

CPLEX Optimizer. 2015. Gurobi Optimizer Reference Manual. (2015). http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud

Dash Optimization. 2005. *Xpress-BCL, Reference Manual* (release 2.6 ed.). Dash optimization. http://www.dashoptimization.com

L. De Alfaro. 1999. *Computing minimum and maximum reachability times in probabilistic systems*. Springer.

A.P. Dempster, N.M. Laird, and Rubin D.B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B* 39 (1977), 1–38.

N.J. Dingle, W.J. Knottenbelt, and T. Suto. 2009. PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets. *SIGMETRICS Perform. Eval. Rev.* 36, 4 (March 2009), 34–39. DOI: http://dx.doi.org/10.1145/1530873.1530881

R. El Abdouni Khayari, R. Sadre, and B. R. Haverkort. 2003. Fitting world-wide web request traces with the EM-algorithm. *Performance Evaluation* 52, 2 (2003), 175–191.

E. A. Feinberg. 2004. Continuous Time Discounted Jump Markov Decision Processes: A Discrete-Event Approach. *Mathematics of Operations Research* 29, 3 (August 2004), 492–524.

A. Gomez, R. Mariño, R. Akhavan-Tabatabaei, A. Medaglia, and J. Mendoza. 2013. A unified framework for vehicle routing problems with stochastic travel and service times. In *Proceedings of TRISTAN VIII*.

Gurobi Optimizaton. 2014. Gurobi Optimizer Reference Manual. (2014). http://www.gurobi.com

N. Hastings. 1971. Technical Note. Bounds on the Gain of a Markov Decision Process. *Operations Research* 19, 1 (1971), 240–244.

B. Heimsund. 2003. *JMP-Sparce Matrix Library in Java*. Technical Report. Univesity of Bergen, Bergen, Norway.

B. Heimsund. 2005. Matrix Toolkits for Java (MTJ). (December 2005). http://rs.cipr.uib.no/mtj/ Last modified: Monday, 05-Dec-2005 09:03:23 CET.

J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. 2005. JAMA: A Java Matrix Package. (July 2005). http://math.nist.gov/javanumerics/jama/ MathWorks and the National Institute of Standards and Technology (NIST).

J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 279–288.

A. Horváth and M. Telek. 2002. Phfit: A general phase-type fitting tool. In *Computer Performance Evaluation: Modelling Techniques and Tools*. Springer, 82–91.

ITEM Software (USA) Inc. 2011. *ITEM TOOLKIT, Version 7*. ITEM Software (USA) Inc. http://www.itemuk.com/assets/docs/ToolKit_Manual.pdf

jMarkov website. 2016. Available online at https://projects.coin-or.org/jMarkov/. (2016).

jUnit. 2016. http://junit.org/. (2016).

W. Knottenbelt. 1996. *Generalised Markovian Analysis of Timed Transitions Systems*. Master's thesis. Department of Computer Science, Faculty of Science, University of Cape Town.

V. Kulkarni. 1995. *Modeling and analysis of stochastic systems*. Chapman & Hall.

M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (LNCS)*, G. Gopalakrishnan and S. Qadeer (Eds.), Vol. 6806. Springer, 585–591.

G. Latouche and V. Ramaswami. 1999. *Introduction to matrix analytic methods in stochastic modeling*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA. xiv+334 pages.

P. L'Ecuyer and E. Buist. 2005. SIMULATION IN JAVA WITH SSJ. In *Proceedings of the 2005 Winter Simulation Conference*.

J. D. C. Little. 1961. A proof for the queueing formula: $L = \lambda W$. *Oper. Res.* 9 (1961), 383–387.

S. Mahadevan, N. Khaleeli, and N. Marchalleck. 1997. Designing Agent controllers using Discrete-Event Markov Models. (1997). Department of Computer Science, Michigan State University.

J. Medina, G. Riaño, and J. Villarreal. 2007. A Dynamic Programming Model for Structuring Mortgage Backed Securities. In *Proceedings of the 2007 IEEE Systems and Information Engineering Design Symposium*, Michael DeVore (Ed.). http://www.sys.virginia.edu/sieds07/papers/SIEDS07_0026_FI.pdf

K. Murphy. 2002. Markov Decision Process (MDP) Toolbox for Matlab. Avilable online at http://www.cs.ubc.ca/~murphyk/Software/MDP/mdp.html/. (2002).

M. Neuts and M. Pagano. 1981. Generating random variates from a distribution of Phase-Type. In *Proceedings of the Winter Simulation Conference*.

M. F. Neuts. 1981. *Matrix-geometric solutions in stochastic models*. The John Hopkins University Press.

M. Olsson. 1998. *The EMpht-programme*. Manual. Chalmers University of Technology, and Götborg University, Available at http://www.maths.lth.se/matstat/staff/asmus/pspapers.html.

T. Osogami and M. Harchol. 2006. Closed form solutions for mapping general distributions to quasi-minimal PH distributions. *Performance Evaluation* 63 (2006), 524–552.

J. F. Pérez and G. Riaño. 2006. 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*. ACM Press, New York.

M. L. Puterman. 1994. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley and Sons, New York.

M. L. Puterman and M. C. Shin. 1978. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science* 24, 11 (1978), 1127–1137.

P. Reinecke, T. Krauss, and K. Wolter. 2012. Hyperstar: Phase-type fitting made easy. In *Quantitative Evaluation of Systems (QEST), 2012 Ninth International Conference on*. IEEE, 201–202.

G. Riaño and J. Góez. 2006. jMarkov: an object-oriented framework for modeling and analyzing Markov chains. In *Proceedings of the SMCtools'06*. ACM Press, New York.

G. Riaño, J. Góez, and J. P. Alvarado. 2006. Modeling Bucket Brigades using Phase-Type Distributions. In *Memorias del XIII Congreso Latino-Iberoamericano de Investigación de Operaciones y Sistemas*. Montevideo, 1 – 6.

G. Riaño. 2002. *Transient behavior of stochastic networks: application to production planning with load dependent lead times*. Ph.D. Dissertation. Georgia Institute of Technology.

A. Riska and E. Smirni. 2007. ETAQA Solutions for Infinite Markov Processes with Repetitive Structure. *INFORMS Journal on Computing* 19, 2 (2007), 215–228. DOI: http://dx.doi.org/10.1287/ijoc.1050.0160

R. Serfozo. 1979. An Equivalence Between Continuous and Discrete Time Markov Decision Processes. *Operations Research* 27 (1979), 616–620.

D. F. Silva, C. Amaya, and G. Riaño. 2009. Continuous-Time Models for Estimating Picker Blocking in Order-Picking-Systems. In *Proceedings of the IIE Conference*.

W. Stewart. 1996. *MARCA: Markov Cahin Analyzer, A software Package for Markov Modelling, Version 3.0*. North Carolina State University, Raleigh, N.C. 27695-8206. http://www.csc.ncsu.edu/faculty/stewart/

Sun Microsystems. 2006. Java Technology. (Jan. 2006). http://www.java.sun.com/

F. Teichteil-Königsbuch, U. Kuter, and G. Infantes. 2010. Incremental plan aggregation for generating policies in MDPs. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 1231–1238.

M. Telek and A. Heindl. 2002. Matching moments for acyclic discrete and continuous Phase-Type distributions of second order. *I.J. of Simulation* 3, 3–4 (2002), 47–57.

A. Thümmler, P. Buchholz, and M. Telek. 2005. A novel approach for fitting probability distributions to real trace data with the EM algorithm. In *Proceedings of the International Conference on Dependable Systems and Networks*.

A. Thummler, P. Buchholz, and M. Telek. 2006. A novel approach for phase-type fitting with the EM algorithm. *Dependable and Secure Computing, IEEE Transactions on* 3, 3 (2006), 245–258.

J. J. Ucrós Ospino. 2012. Application of phase-type distribution in cycle time estimation of queueing networks : a case of semiconductor manufacturing systems. (2012). Master thesis, Universidad de los Andes.

P. Zipkin. 2000. *Foundations of Inventory Management*. Mc Graw Hill.