QBD Sensitivity Analysis Tool Using Discrete-Event Simulation and extension of *SMCSolver*

M. Cordy
The University of Namur
Faculty of Computer Science
Rue Grandgagnage, 21
5000 Namur, Belgium
maxime.cordy@fundp.ac.be

M.-A. Remiche
The University of Namur
Faculty of Computer Science
Rue Grandgagnage, 21
5000 Namur, Belgium
mre@info.fundp.ac.be

ABSTRACT

We propose a tool that provides both analytical and simulation based performance analysis of both homogeneous and inhomogeneous Quasi-Birth-and-Death (QBD) processes. We extend SMCSolvers in order to study inhomogeneous case and to analyze first passage times. Simulations are performed on a discrete-event based approach. We also provide a rich input interface to give the most flexibility to the user to define its QBD transitions. The analysis of sensitivity in a complex level-dependent QBD model of a reliable system is an illustration of the wide range of QBD the tool may help to analyze.

1. INTRODUCTION

In the past, Quasi-Birth-and-Death (or QBD in short) processes have been extensively used for the design and the performance analysis of a great variety of systems, such as reliability systems (see [13] for example), peer-to-peer systems (see [5]), fluid Markov models (see [3] among many others) or call center systems (see [7]) to cite but a few. QBD processes are particular cases of Markov processes, defined on state space with the following structure

$$\mathcal{S} = \{(k,i); k \in \mathbb{N}, 0 \le i \le n_k\}, \tag{1}$$

with usually $n_k < \infty$ for all $k \in \mathbb{N}$. Accordingly, their generator has the following form

$$Q = \begin{pmatrix} B_0 & B_1 & 0 & 0 & \dots \\ A_{-1}^{(1)} & A_0^{(1)} & A_1^{(1)} & 0 & \dots \\ 0 & A_{-1}^{(2)} & A_0^{(2)} & A_1^{(2)} & \dots \\ 0 & 0 & A_{-1}^{(3)} & A_0^{(3)} & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}, \tag{2}$$

where the inner blocks $A_i^{(k)}$ are of size $n_k \times n_{k+i}$, for $k \in \mathbb{N}_0$ and $i \in \{-1,0,1\}$, and B_i is of size $n_0 \times n_i$, $i \in \{0,1\}$. Diagonal elements of B_0 and of $A_0^{(k)}$, for all $k \in \mathbb{N}$ are negative.

VALUETOOLS 2011, May 16-20, Paris, France
Copyright © 2011 ICST 978-1-936968-09-1
DOI 10.4108/icst.valuetools.2011.245745

Other elements are positive or null. Moreover, we have

$$B_0\vec{1} + B_1\vec{1} = \vec{0} \tag{3}$$

$$A_{-1}^{(k)}\vec{1} + A_0^{(k)}\vec{1} + A_1^{(k)}\vec{1} = \vec{0} \tag{4}$$

for all $k \in \mathbb{N}$, where $\vec{1}$ and $\vec{0}$ respectively are vectors full of 1 and 0 respectively and of appropriate size.

When $A_i^{(k)} = A_i$ for all $k \in \mathbb{N}_0$ and for all $i \in \{-1,0,1\}$ (expect for $A_{-1}^{(1)}$), the QBD is said to be *homogeneous*. Otherwise, the QBD is said *inhomogeneous*. In case there exists $K < \infty$, such that

$$Q = \begin{pmatrix} B_0 & B_1 & 0 & \dots & & \\ A_{-1}^{(1)} & A_0^{(1)} & A_1^{(1)} & \dots & 0 & 0 \\ 0 & A_{-1}^{(2)} & A_0^{(2)} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & B_{-1} & B_K \end{pmatrix}, \tag{5}$$

with B_{-1} of size $n_K \times n_{K-1}$ and B_K of size $n_K \times n_K$; the QBD is said to be *finite*. In case it does not exists such a K, the QBD is said *infinite*.

We refer to Latouche and Ramaswami [8] for a clear introduction to the matrix analytic methods that exist to perform QBDs analysis.

To perform sensitivity analysis of such systems is of great interest, in particular when measuring the robustness of optimal designed policies. Our aim is to propose a tool that would, as a first objective, rapidly give some insight about the sensitivity of QBD models subject to small variations on their input parameters. Our effort will be put on developing such a tool that will use either simulation or exact methods to solve the QBD, depending on the nature of the performance problem itself.

At this stage, exact computations are performed on both the original and perturbated QBD, except in the case of a $M/PH/1$ queue where we used results developed by Dendievel et al. in [4]. The tool also proposes to make use of a discrete-event based simulation procedure. This is of particular help when the size of the QBD is large. The tool supports the analysis of both homogenous and inhomogeneous QBDs. Input parameters can be specified following three different formats : the user may specify (i) the type of queue using Kendall's notation, (ii) the complete generator,

(iii) the possible transitions.

There exists different tools to solve QBD, see for example MAMSolver developed by Riska and Smirni in [14], or SMC-Solver proposed in [2]. We decide to incorporate the second tool to solve exactly homogenous QBD and to extend it to the analysis of inhomogeneous QBD. We also propose to add a function to compute first passage times.

Such tools usually propose to specify the blocks that compose the QBD generator (see Equation (2)). As mentioned earlier, our tool allows the user to specify the input parameters by means of transition specification. This method is indeed necessary to implement in order to offer the possibility to deal with general structured inhomogeneous QBDs. In Katoen et al. [6], all possible transitions have to be given by the user. In our tool, repetitive structure of the inner blocks can be defined as such.

The paper is composed of three main sections. First, we explain the tool design choices operated for both the input and output interfaces and the architecture of the code itself. In the following section, we perform as an illustration to our work, a sensitivity analysis of a preventive repair policy in a reliable system. This model was first discussed in [12]. Finally, we conclude our work by indicating work that need to be done in the future.

2. TOOL ARCHITECTURE

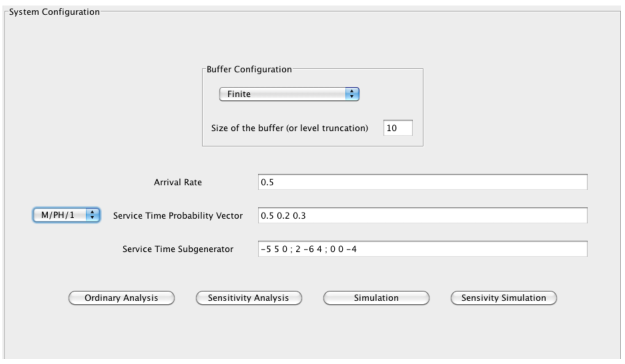
In this section, we discuss the specificity of the tool regarding the input and output interfaces, as well as the architecture of the code itself.

2.1 Input interface

An interesting feature of the tool is the ability for the user to specify the QBD generator by using one of the three different input interfaces. We now describe each method.

The first possibility is to define the QBD as a queueing system, using Kendall's notation in concise form. After choosing the queue type, each parameter of the selected system, such as an exponential rate or a phase-type distribution, must be entered. We refer to Latouche and Ramaswami [8], Chapter 2 for a clear introduction to phase-type distributions. At the moment, the tool is restricted to the $M/M/1$, $M/PH/1$, $PH/M/1$ and $PH/PH/1$ queues. However, it can be extended to be able to work with more queue types. The size of the buffer, possibly infinite, can also be chosen. Figure 1 illustrates how our input interface looks like. In this example, we want to specify an M/PH/1 queue. We must define the size of the buffer, the arrival rate, the probability row vector and the generator matrix of the phase-type distribution. In this particular example, the generator is a 3×3 matrix, where each component of a row is separated by a comma and each line of the matrix is separated by a semicolon.

The second input interface allows to explicitly define the blocks that compose the generator. For this purpose, the user must provide a Matlab or Octave function. **Octave** is a programming language specialized in numerical computations. Its syntax and its semantic are almost identical to the


System Configuration

Buffer Configuration

Finite

Size of the buffer (or level truncation): 10

Arrival Rate: 0.5

M/PH/1 — Service Time Probability Vector: 0.5 0.2 0.3

Service Time Subgenerator: -5 5 0 ; 2 -6 4 ; 0 0 -4

Figure 1: Definition of a QBD generator as a queueing system, using Kendall's notation.

MATLAB ones. Octave is a free software under the terms of the GNU General Public License.

This function would have one parameter: that is i, the level for which the inner blocks would be computed. The output parameters of the function are then the three inner blocks corresponding to this level, given in the following order: $A_{-1}^{(i)}$, $A_0^{(i)}$ and $A_1^{(i)}$. If one of these three matrices is not defined for the considered level, the function returns an empty matrix. For example, the result of the function for level 0 would be: [], B_0, B_1 as specified in Equation (2).

For the program to load the function, the user has to provide the path where to find it. Once the function loaded, the tool has everything it needs to define the QBD process. The main drawbacks of this method are that first programming in Octave is required. Second, knowing the exact and complete form of the generator is required, while it may be easier to define only the transitions of the corresponding process. The next and last method is an answer to these problems.

The third and last input method consists in specifying the possible transitions of the process in a syntax consistent and textual language we developed. The particular feature of our method is that all the possible transitions have not to be given. Instead, the user may specify more concisely a repetitive structure appearing in the generator. Our tool is now restricted to two-dimensional Markovian state spaces. A small context-free grammar has thus been developed in order to build our parser. It reads a text file respecting this grammar and accordingly, produces an Octave function. As for the second method, this function returns the inner blocks of a given level. For readability reason, we present in the appendix the most important symbols of the grammar in Backus-Naur Form, as well as their semantic.

Figure 2 illustrates this specification method for the definition of an M/PH/1 queue. The arrival rate is *lambda*. The service time distribution is completely specified using four parameters, these are *mu1, mu2, p* and *q*. As specified in our code, these input parameters are *constant* in the specification of the transitions.

We are then able to specify the *transitions*. Each transition definition begins with the keyword `TRANSITION`, followed by

```
CONST lambda = 0.5;
CONST mu1 = 2;
CONST mu2 = 3;
CONST p = 0.6;
CONST q = 0.4;

TRANSITION (SAME,1)      FOR L = 0 AND P = 1             RATE -lambda;
TRANSITION (UP,1)        FOR L = 0 AND P = 1             RATE lambda;

TRANSITION (DOWN,1)      FOR L = 1 AND P = 1             RATE mu1 * p;

TRANSITION (SAME,P+1)    FOR L > 0 AND P = 1             RATE mu1 * q;
TRANSITION (DOWN, 1)     FOR L > 1 AND P = 1             RATE mu1 * p;
TRANSITION (DOWN, 1)     FOR L > 1 AND P = 2             RATE mu2;
TRANSITION (UP,P)        FOR L > 0 AND P > 0 AND P < 3   RATE lambda;
```

Figure 2: Definition of an M/PH/1 queue using text-based transition specification.

the type of transition and the destination phase. More precisely,

- possible type of transitions are `UP`, `DOWN` or `SAME` respectively, which describes movement to the upper level, the downer level or in the same level, respectively and
- the phase is explicit or is given through some constraint.

We then give these constraints for the level `L` and the phase `P`. The instruction finishes when the *rate* is finally given using possible predefined constants.

In Figure 2, we observe that the first specified transition means that when in level 0 and phase 1, the process may stay is that state for an exponential amount of time whose rate is `lambda`. We also observe that the fourth specified transition concerns all the level strictly greater than 0, but only phase 1. It indicates that in that starting state, the process may move to state `(L,P+1)` with a rate of probability `mu1` $\times$ `q`.

Once the QBD is defined, the tool may analyze it using matrix analytic methods, or may simulate it via discrete-event simulation and accordingly, carry out a sensitivity analysis. The user indicates his choice by clicking on the corresponding button. When simulating it, some additional parameters must be set, such as the starting level, the starting phase and the simulation end time.

When doing sensitivity analysis, the tool will work with a perturbed generator defined as follow:

$$Q_{pert} = Q + \epsilon\tilde{Q} \tag{6}$$

Firstly, the perturbation generator $\tilde{Q}$ must be specified via the same input interface that was used to define the QBD generator Q. After that, a set of values that ϵ will take must be chosen. Then, for each selected value of ϵ, the tool will perform an analysis or a simulation on the resulting Q_{pert}, whose inner blocks will be computed dynamically.

2.2 Output interface and performed analysis

The tool computes different data that may be useful when evaluating the performance of a QBD process. In this first version of the tool, we focus on the stationary probability vector $\vec{\pi}$ and first passage times.

Following the matrix analytic methods, key matrices R_i (with i being the level) must be computed in order to determine the stationary probability vector. In the particular case of an homogeneous QBD, we have used the SMCSolver ([2]) implementation of the *Logarithmic Reduction* algorithm (see [8], Chapter 8), which is quadratically convergent. To handle the level-dependent case, we have implemented in Octave the algorithm presented in [8], Chapter 12.

In [8], Chapter 11, two algorithms to compute the expected first passage time from the level 0 to any upper level of an homogeneous QBD are detailed: these are the *Linear Level Reduction* and the *Reduction by Bisection* algorithms. We have chosen to implement the former, as it can be more easily generalized to compute the expected first passage time from any level to any other one, in both the homogeneous and inhomogeneous cases. Our tool implements that generalization.

In case of a simulation, we use a discrete-event simulation approach (see Leemis and Parkin [9]). The main lines of the algorithm are as follows, assuming the process is in level i, phase ϕ,

- the inner blocks $A^{(i)}_{-1}, A^{(i)}_{0}, A^{(i)}_{1}$ corresponding to the current level i are obtained.
- These blocks are discretized with a rate r equal to the maximum absolute value of the diagonal of $A^{(i)}_0$. We follow here the principles of the uniformization method (as described in Latouche and Ramaswami [8], Section 2.8).
- Next only the line corresponding to the current phase (ϕ) is considered. It now gives the transition probabilities that after a random exponential time t (with rate r), the process moves to another state or remains in state (i, ϕ). We simulate that transition.
- Accordingly, the process moves to this state and the simulation time is increased by t.

We repeat this procedure until the simulation end time is reached.

Note that the algorithm covers both the homogeneous and inhomogeneous cases. However, for the homogeneous case, an optimization can be made by computing only one time the inner blocks as those are identical for each level, except for the levels 0, 1 and the two last levels in case of a finite QBD.

The stationary probability of a given state is then estimated by dividing the time spent in this state during the simulation by the total simulation time. We also provide the confidence interval round our estimation (see Figure 3 for one particular M/PH/1 case).

First passage times are obtained as arithmetic mean of time at which the simulation reached the level of interest. Enough

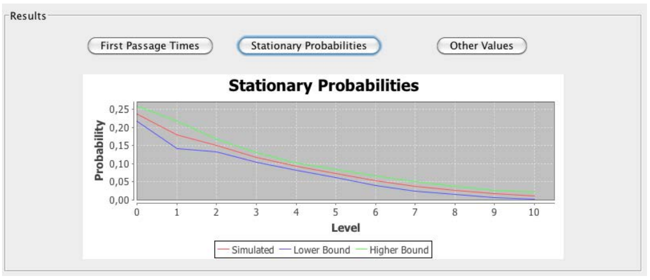


Figure 3: Simulation results for the stationary probability of a M/PH/1 case.

simulations must be performed and confidence intervals are always provided to measure it.

The outputs are given either in graphical format or as a text file. The text file presents at each line the steady state probabilities of the corresponding level. When a plot is provided, Xaxis gives the level while Yaxis gives the probability that the process is in level i, whatever the phase (see Figure 3 for an example of the graphical output interface). Finally, the expected first passage time starting in a given level i can also be plotted in a graph whose abscissa is the target level j and the ordinate is the expected first passage time from i to j. The expected first passage times from i can also be given via in a text file. In this case, for each destination level j, we provide the expected first passage time to that level for each possible phase of the starting level i.

As mentioned before, the tool allows to do some sensitivity analysis either by using matrix analytic methods or simulations on the perturbed process. The same data can then be computed for each ϵ value of the perturbation. Then, a graph showing the influence of the perturbation on a specific performance measure (that is the stationary probability of a particular level or the first passage time from a given level to another one) can be displayed. The Xaxis gives the different values ϵ takes. The Yaxis is the value of the considered performance measure.

2.3 Code architecture

The tool is composed of some Java and GNU Octave modules. One of our goals is to make it quite easily modifiable. Therefore, we focus on modularity by dividing the program into high-level components. Figure 4 shows those different components and the dependencies between each other. Each green box represents a component. The dependencies are indicated by the arrows. The component at the origin of an arrow depends on the component at the edge of this arrow. Before describing each component, its roles and dependencies, we first motivate the choice of the programming language.

Each **.m** file contains all the Octave functions implementing the matrix-analytic methods and simulation algorithms. Some of them come from SMCSolver [2]. We choose to provide these functions as a stand-alone resource so that one may be able to call them directly with Octave, instead of only observing their results via the graphic user interface developed in Java.

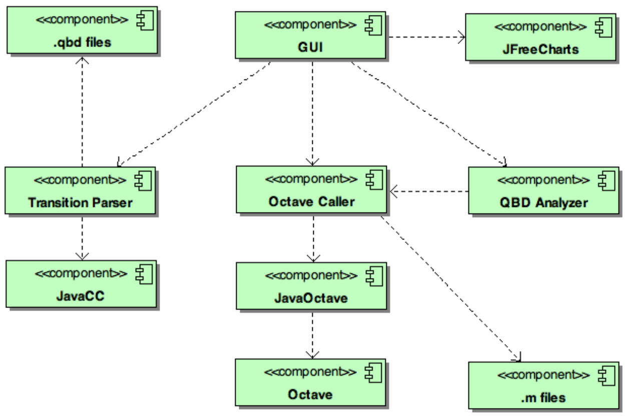


Figure 4: High-level architecture of the tool.

We choose to use **Octave** as the computation core of our tool because of its higher performance in numerical (mainly matrix) computation, compared to Java. Two features of this language must be kept in mind:

- Octave uses an interpreter to execute instructions which are written through its command-line interface or are contained in a script.
- It allows to dynamically load some new functions or to redefine an existing one.

JavaOctave is a Java library that works as a bridge from Java to Octave. It was developed by Kim Hansen[1]. It allows to call the Octave interpreter from a Java program and to transform an Octave data structure into a Java object. Basically, it runs Octave and provides some functions to send back the instructions written in the Octave language.

Let us now present some of the main components in Figure 4.

OctaveCaller is a Java component we developed. It is the only component that communicates with JavaOctave. Thanks to this property, modularity is improved: the other Java components are completely independent from Octave components. Since JavaOctave is a bridge between Java and Octave, OctaveCaller can be seen as a gate on the Java side. It provides routines to keep track of every Octave function that has been declared. Any function can thus be called and its return values will be contained in Java objects managed by the OctaveCaller.

The Java component **QBD Analyser** is the masterpiece of the tool. It determines which function must be called, depending on the chosen analysis method, the characteristic of the defined QBD process and the desired output data. It determines which function has to be registered in the OctaveCaller, when and how such a function must be modified. It also sets the right parameters of a function and orders the OctaveCaller to call it. Then, it analyzes the data structures returned by the OctaveCaller to extract the desired results.

[1]http://kenai.com/projects/javaoctave/

The general performance of the tool highly depends on the choices of the QBD Analyzer. For example, when a simulation of an homogeneous QBD is required, it can either choose to call the general function that can simulate any QBD or the specific function for the level-independent case. The latter is far more efficient than the former, as the inner blocks of the QBD are computed once at the start of the simulation instead of being determined at each level movement.

JavaCC[2] is an open source tool originally owned by Sun Microsystems. Basically, it allows to generate a lexical analyzer and a descending syntax analyzer from a conflict-free grammar defined in Backus-Naur Form (see [1] Chapters 1 to 4, for a clear introduction). A FAQ[3] about JavaCC is also maintained by Theo Norvell at Memorial University of Newfoundland.

We used JavaCC to create the **Transition Parser** component. Its role is to parse a text file (a **.qbd** file) respecting the grammar we developed (see Section 2.1 and Appendix A) and to produce an Octave function that dynamically computes the inner blocks of the QBD corresponding to the defined transitions. Accordingly, the input grammar and the output language are completely independent.

JFreeChart[4] is a free library that allows the developers to display graphs inside their Java applications. It is distributed under the GNU Lesser General Public License. It supports a large variety of graph types and provides a complete API for dynamically editing the graph, as well as performing zooms on it.

Finally, the **Graphic User Interface (GUI)** component has two main roles. Firstly, it displays the different windows through which the user can navigate. That includes both the input windows, in which the parameters of the studied system are entered and the path to essential files, as well as the output windows in which the evaluation results are displayed. Secondly, it manages the user actions and requests the right service of the right component when necessary. Basically, this component reads the data introduced by the user, calls the right input interface in order to set every parameter and ask the QBD Analyser to perform the analysis.

3. SENSITIVITY ANALYSIS OF A RELIABLE SYSTEM

As an illustration to our tool, we propose to analyze the sensitivity of a particular reliable system, as first defined and analyzed in [12]. We choose this model as it constitutes one clear example of the use of inhomogeneous finite QBD (see Equation (5)) to model a complex system. It thus allows us to clearly highlight the use and the versatility of our tool.

In this section, we explain the system itself. We then clarify the state-space's definition as exposed in [12]. We propose in this summary, to give the inner structure of the $A_0^{(i)}$, $1 \leq i \leq n-1$ only. We choose the second method to specify the structure of the generator to the tool. Finally, we propose to measure the sensitivity of the system subject to longer or shorter inspection.

[2]https://javacc.dev.java.net/

[3]http://www.engr.mun.ca/∼theo/JavaCC-FAQ/

[4]http://www.jfree.org/jfreechart/

System definition

The system is composed of n units. One unit is online, the others are in warm standby. However units are subject to degradation and eventually may go to *corrective* repair. Only one unit may be repaired at a time, others are queueing in FIFO order. To prevent full degradation when online, inspections are randomly performed. In case the level of degradation is too high, a *preventive* repair is performed, except if the system contains no more standby unit. Would the system be empty of standby units and the online unit need a corrective repair, one unit in preventive repair would be preempted (if available). In the other case, the system would be said to have failed.

Standby and online lifetimes (prior to full degradation) respectively are assumed to be phase-type distributed PH(α_s, T_s) of order m_s and PH(α, T) of order m respectively. Preventive and corrective repairs take a random phase-type distributed time, with parameter (β_p, S_p) of order n_p and (β_c, S_c) of order n respectively. Inspection times are random and two consecutive inspection procedures are separated by a phase-type random time with parameters (γ, L) of order ν. All random variables are independent of each other.

State-space definition and decomposition

As estabished in [12], the system can be modeled as a Markov process whose state-space is

$$(i, j, k, z^{(i,j)}, l_p, l_c, f) \tag{7}$$

where

- i is the number of units in preventive repair, with $0 \leq i \leq n-1$,
- j is the number of units in corrective repair, with $0 \leq j \leq n$,
- k is the phase occupied by the online unit, with $1 \leq k \leq m$,
- $z^{(i,j)}$ is the phase of the standby units,
- l_p is the phase of the unit in preventive repair, with $1 \leq l_p \leq n_p$,
- l_c is the phase of the unit in corrective repair, with $1 \leq l_c \leq n_c$,
- f is the phase of the inspection procedure, with $1 \leq f \leq \nu$.

Let us be more precise about $z^{(i,j)}$. This vector is composed of $(z_1, z_2, \ldots, z_{n-1-(i+j)})$, with $1 \leq z_r \leq m_s$, where $1 \leq r \leq n-1-(i+j)$.

The *level* M is defined as the number of units in repair (either preventive or corrective repair), that is $0 \leq M \leq n$, with

$$M = \{(i, j); 0 \leq i \leq n-1 \text{ and } j = M - i\}, \tag{8}$$

$$M = \{(0, n)\}. \tag{9}$$

We now explain the inner structure of one particular block, that is $A_0^{(M)}$, for $2 \leq M \leq n-2$. Other blocks that compose generator (5) are clearly defined in Appendix A of [12]. Our objective is here to illustrate the complexity of the inhomogeneous QBD that may be handled by our tool.

We have

$$A_0^{(M)} = \begin{pmatrix} A_0^{(M)}(1,1) & & \\ & A_0^{(M)}(2,2) & \\ & & A_0^{(M)}(3,3) \end{pmatrix}, \qquad (10)$$

where level M has been partitioned in three subsets, that is $M = M_1 \cup M_2 \cup M_3$, defined as follows

$$M_1 = \{(0, M)\} \qquad (11)$$

$$M_2 = \{(i,j); 1 \leq i \leq M-1, j = M-i\} \qquad (12)$$

$$M_3 = \{(M, 0)\}. \qquad (13)$$

This explains why only diagonal blocks are non-null. Indeed, moving from a state in M_i to a state in M_j $(j \neq i)$ implies some repair to be finished and a new one to start.

We then have for matrix $A_0^{(M)}(1,1)$

$$\begin{aligned} A_0^{(M)}(1,1) &= T \otimes I_{(m_s)^{n-M-1} n_c \nu} \\ &+ I_m \otimes (T_s \oplus \ldots \oplus T_s) \otimes I_{n_c \nu} \\ &+ I_{m(m_s)^{n-M-1}} \otimes S_c \otimes I_\nu \\ &+ I_{m(m_s)^{n-M-1} n_c} \otimes L \\ &+ U_1 \otimes I_{(m_s)^{n-M-1} n_c} \otimes L^0 \gamma, \end{aligned} \qquad (14)$$

where I_n is the identity matrix of size n,

$$U_1 = \begin{pmatrix} I_g & 0 \\ 0 & 0 \end{pmatrix}_{m \times m} \qquad (15)$$

is a matrix that permits to identify the states in which the units do not need to go to corrective repair (the first g phases are ok, others are not), and

$$L^0 = -L\,\vec{1}. \qquad (16)$$

Equation (14) is readily explained as follow. No change of level in this Markov process implies that we did observe a change of phase only. This can be a change of phase for the online unit (determined by T) or ("+") for one of the standby units (determined by T_s) or ("+") for the corrective repair unit (determined by S_c) or ("+") for the inspection unit (determined by L). In case a new inspection period starts (determined by $L^0\gamma$), the online unit was not in a too degraded state (determined by U_1).

In our tool, we choose the second input interface to specify these matrices. This one example showed the complexity of this inhomogeneous QBD. The best is to code all the matrices according to an Octave program and to let our program load it and call it when necessary.

Sensitivity analysis

We propose to identify the impact of a shorter or longer inspection period on the rocof of the system, that is the probability that the system fails completely (i.e. no more unit is available to become the online unit).

Authors in [12] had proposed such a study based on the phase g (see U_1 definition in (15)) at which the unit needed to go to preventive repair. They were able to measure the prize of being more strict on the need to go preventive repair. We wish here to see if performing more often inspection could have the same effect on the rocof of the system. For this we choose a perturbation of the inspection procedure as follows

$$L_\epsilon = L + \epsilon A \qquad (17)$$

where

$$A = \begin{pmatrix} -1 & 1 \\ 0 & -1 \end{pmatrix}, \qquad (18)$$

and ϵ ranges from 0.01 to 0.09 by step of 0.02.

We choose exactly the same input parameters as they did in [12] and obtain the results in Table 1.

We have chosen ϵ such that the greater ϵ, the smaller the interval in between two consecutive inspections. Accordingly, we observe in Table 1 that for a given g, the greater ϵ, the smaller the rocof. Indeed, the system will repair more rapidly the default units. As established in [12] the greater g the greater the rocof. This makes sense since on the contrary in this case, the inspection will cause a preventive repair more lately. With this sensitivity analysis, we may now decide about a compromise in between the phase of decision for preventive repair (that is g) and the rate of inspection, that is

$$\mu = \gamma(-(L + \epsilon A))^{-1}\vec{1}. \qquad (19)$$

4. PERSPECTIVE AND FUTURE WORK

There are two main directions we wish to follow in the future to extend our tool. First, we wish to integrate recent and further developments on sensitivity analysis of QBD process. Second, the grammar need to allow the user to define more complex QBD transition structure and state space.

Sensitivity analysis is the subject of many research papers (see for example [10] or [11]) but few propose a systematic and tractable approach to any kind of QBD. In our tool, at this stage of the development, we have chosen to carry out two type of analysis, one on the original QBD and one on the perturbated QBD. This is clearly not efficient and our tool will definitely need to integrate recent advances in this matter.

Future work on the grammar should permit to extend our approach to n-dimensional Markovian state-space. We should also be able to cover the case where complex dependencies in between state transition and rate of transition occur.

Let us conclude that recent developments on QBD simulations techniques (such as perfect simulation for example) might be of great interest to be included in our tool.

APPENDIX

A. DEFINITION OF THE GRAMMAR USED FOR THE INPUT INTERFACE

We develop a small context-free grammar in order to build a parser that reads a text file respecting this grammar and

Table 1: Performance analysis on the rocof of the system

$g/\ \epsilon$	0.01	0.03	0.05	0.07	0.09
1	1.4921e-06	8.3196e-07	6.3283e-07	5.4581e-07	4.9886e-07
2	2.4186e-06	1.5270e-06	1.2121e-06	1.0623e-06	9.7770e-07
3	5.9177e-06	4.4985e-06	3.8340e-06	3.4591e-06	3.2224e-06
4	1.5043e-05	1.4081e-05	1.3302e-05	1.2666e-05	1.2141e-05

constructs the generator of a QBD process. This appendix presents all the symbols composing the grammar as well as their informal semantic. It also aims to explain how to specify the transitions of a QBD thanks to them. Let us recall that our approach is limited to two-dimensional Markovian state spaces. All the grammar symbols are defined in figure 5.

Our QBD specification method consists in the declaration of a number of constants followed by the declaration of a number of transitions. We propose to consider Figure 2 as a clear example of the use of the grammar. While it is allowed not to declare any constant, we have imposed the restriction that at least one transition must be defined.

The declaration of a *constant* begins with the keyword `CONST`, which is followed by an *id* and then a *value*. Variable *id* simply represents the name of the constant. It is a string beginning with a lower case letter. This letter can be followed by a series of some of the following characters: letters, digits, underscore. A *value* is a decimal number and represents the value of the constant.

The definition of a transition begins with the keyword `TRANSITION`, followed by the type of transition and the destination phase. More precisely, the possible types of transition are `UP`, `DOWN` and `SAME` respectively, that describes if the process moves to the upper level, the downer level or stays in the same level, respectively. The destination phase has either explicit value or depends on the starting phase P.

Then, a series of constraints on the starting level L and starting phase P are specified. They are preceded by the keyword `FOR` and separated from each other by the keyword `AND`. The conjunction of these constraints explains for which source states the transition is defined. Finally, the keyword `RATE` is followed by an expression giving the rate at which the transition occurs. This can be a simple expression such as a decimal number, an *id* referencing a constant. It can also be compound, i.e. simple expressions joined by an arithmetic operator ($+, -, *$ and $/$) or by the binary function `min` and `max`. Finally, expressions can be grouped using parentheses.

Accordingly, one could define the next transition using our grammar by writing:

```
TRANSITION (UP,1) FOR L > 1 AND P = 3 RATE 5.2
```

It means that the process can move from every state (L, P) such that $L > 1$ and $P = 3$ to the upper level and in phase 1, thus to the state $(L + 1, 1)$ with rate 5.2.

B. THE OCTAVE FUNCTIONS

```
qbd          ::=  (constant)* (transition)+
constant     ::=  CONST id value
transition   ::=  TRANSITION (move, expression)
                  FOR conditions RATE expression
move         ::=  UP | SAME | DOWN
conditions   ::=  condition (AND condition)*
condition    ::=  (L | P) (< | <= | = | >= | >) expression
expression   ::=  term term_0
term_0       ::=  ((+ | -) expression) | ε
term         ::=  factor factor_0
factor_0     ::=  ((* | /) term) | ε
factor       ::=  id | L | P | value
                  | ( expression )
                  | - expression
                  | min( expression , expression )
                  | max( expression , expression )
id           ::=  [a-z]([a-z] | [A-Z] | [0-9] | _])*
value        ::=  (0 | [1-9] [0-9]*) ([.][1-9][0-9]*)?
```

Figure 5: Definition of the symbols of the context-free grammar. ϵ represents the empty symbol.

We use Octave as the computation core of our tool. Therefore, we provide some Octave functions that implement the simulation and the matrix-analytic algorithms. These functions can be either called from the Octave command line or from our Java code. In this appendix, each function is described. We explain specifically what are their parameters and their return values.

The functions can be separated into two groups. The first group is a set of functions implementing the matrix-analytic algorithms to compute the rate matrices, the stationary probability vector or the expected first passage times. The second group includes the simulation functions, as well as other functions used to provide some statistics from the simulation results.

Matrix-analytic functions

One of our goal is to compute the stationary probability vector. In the infinite and homogeneous case, we use some functions of the SMCSolver (see [2]): **QBD_LR.m** and **QBD _pi.m**. The first one computes the rate matrix R by using the *Logarithmic Reduction* algorithm. The second one computes the steady state vector. In case of finite and inhomogeneous QBDs respectively, we define two other functions: **QBD_pi_finite.m** and **QBD_pi_Inh.m** respectively. Both implement a modified version of the *Linear Level Reduction* algorithm ([8], Chapters 10 and 12). One key difference beween these two functions is how they get the inner blocks. In the former function, they are passed as parameters. Thus, the arguments of **QBD_pi_finite.m** are: $A_{-1}^{(1)}$, B_0, B_1, A_{-1}, A_0, A_1, $A_{-1}^{(K)}$, $A_0^{(K)}$, $A_1^{(K-1)}$ and K (as defined in Equation (5)). In the latter case, that is **QBD_pi_Inh.m**, the inner blocks are dynamically computed by calling another function that returns those and that takes only one parameter:

the level to compute the block of. This function must be implemented in the file **computeLevelMatrices.m**.

Once the steady state vector is computed, it can be passed as a parameter of **QBD_stat.m**. This function returns the mean stationary visited level and its standard deviation.

To compute the expected first passage times from a given level to an upper one, an algorithm was proposed in [8], Chapter 11. The function **QBD_fpt_LLR.m** implements a generalized version of this algorithm. It allows to compute the expected first passage times from every phase of a given level to any another level. It takes two parameters: s, the origin level and d, the destination level. The inner blocks are dynamically computed whenever they are needed thanks to **computeLevelMatrices.m**.

Simulation functions

The discrete-event simulation algorithm is implemented via three functions. Again, these functions only differ in the way the inner blocks are obtained.

QBD_sim_hom.m simulates an infinite homogeneous QBD. Six inner blocks are needed: $A_{-1}^{(1)}$, B_0, B_1, A_{-1}, A_0 and A_1. They are computed and stored before the simulation begins.

To simulate an homogeneous and finite QBD, three more inner blocks are needed: B_{-1}, B_K, $A_1^{(K-1)}$ (if different from A_1). Along with the six previously defined blocks, they can be passed as parameters of the function **QBD_sim_hom _fin.m**.

Finally, the inhomogeneous case is simulated by the function **QBD_sim_Inh.m**. The inner blocks of a given level are computed when this level is reached for the first time. Then, these blocks are stored in order to be used whenever they are needed. Thus, a block is computed at most one time.

Each simulation function computes the estimated stationary probability of every state of the process, the mean visited level, its standard deviation, the lowest and the highest reached levels.

The first passage time from the starting level to any reached level is also returned as a vector.

If the number of batches is greater than one, a confidence interval for the expected probability of every state is also computed. The lower (respectively higher) bounds are contained in the returned value *lowerBounds* (respectively *higherBounds*).

This interval is obtained by calling **Estimate_Mean.m**. This function has two parameters: *Sample*, a vector that contains the sample values, and *alpha*, the level of confidence of the estimation.

C. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] D. Bini, B. Meini, S. Steffe, and B. Van Houdt. Structured markov chains solver : software tools. In *Proceedings of SMCTOOLS '06.* ACM Press, 2006.

[3] A. da Silva Soares and G. Latouche. Matrix-analytic methods for fluid queues with finite buffers. *Performance Evaluation*, 63:295–314, 2006.

[4] S. Dendievel, G. Latouche, and M.-A. Remiche. Perturbation analysis of an M/PH/1 queue. In *Performance 2010, posters*, 2010.

[5] S. Hautphenne, K. Leibnitz, and M.-A. Remiche. Modeling of P2P file sharing with a level-dependent QBD process. In W. Yue, Y. Takahashi, and H. Takagi, editors, *Advances in Queueing Theory and Network Applications*, pages 247–263. Springer, 2009.

[6] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244, Los Alamos, CA, USA, 2005. IEEE Computer Society.

[7] K. Kawanishi. QBD approximations of a call center queueing model with general impatience distribution. *Computers & Operations Research*, 35:2463–2481, 2008.

[8] G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling.* ASA-SIAM Series on Statistics and Applied Probability. SIAM, 1999.

[9] L. Leemis and S. Park. *Discrete-event Simulation. A First Course.* Pearson, 2006.

[10] Q.-L. Li and L. Liu. An algorithmic approach for sensitivity analysis of perturbed quasi-birth-and-death processes. *Queueing Systems*, 48:365–397, 2004.

[11] C. D. Meyer. Sensitivity of the stationary distribution of a Markov chain. *SIAM J. Matrix Anal. Appl.*, 15:715–728, 1994.

[12] D. Montoro Cazorla and R. Pérez-Ocón. An ldqbd process under degradation, inspection, and two types of repair. *European Journal of Operational Research*, 190(2):494–508, 2008.

[13] R. Pérez-Ocón and D. Montoro-Cazorla. A multiple system governed by a quasi-birth-and-death process. *Reliability Engineering and System Safety*, 84:187–196, 2004.

[14] A. Riska and E. Smirni. Mamsolver: A matrix analytic methods tool. In *TOOLS '02: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 205–211, London, UK, 2002. Springer-Verlag.