INTRODUCTION TO PERFORMANCE EVALUATION OF SYSTEMS

Ricardo M. Czekster, Thais Webber
Performanceware Technologies
Rua Dr. Flores, 262, CEP 90020-120
Porto Alegre, Rio Grande do Sul, Brazil
{rczekster, webber.thais}@gmail.com

Latest update: April 10, 2020

ABSTRACT

This is an ongoing effort to define quantitative *Performance Evaluation* (PE), written to clarify the understanding of some concepts on terminology thus avoiding ambiguity in daily efforts when investigating systems characteristics. For a glossary of terms and definitions provided by standard bodies, it is highly suggested to peruse extensive information offered by IEEE or ISO in [1, 2] The intention of this work is to select important notions applicable to PE and present means and approaches suitable for analysis and decision making.

***K*eywords** Performance Evaluation · Taxonomy · Techniques

1 Introduction

The world is fundamentally composed of a magnitude of different systems and components, for example, cars, trains, machines, computers, *Cyber-Physical Systems* (CPS), *Internet-of-Things* (IoT) devices, and so on. Systems are not usually evaluated in isolation because they are coupled together in a *System-of-Systems* (SoS) network of hyper-connected devices (sensors, routers, servers, and many other). The research area of *Quantitative Performance Evaluation of Systems* (PE) emerges as a valid approach to tackle a myriad of problems in today's society. This brief introductory material aims to discuss key aspects of systems with focus on performance [3], among other characteristics.

Systems today (and for the foreseeable future) possess a high degree of limited capabilities in terms of processing power, energy and storage requirements as well as reduced mobility or special needs (e.g. interoperability with other technologies and so on). Examples are sensors, wearable computing, IoT devices, electrical vehicles, cell phones and tablets. Amidst all those systems, telecommunication networks and its underlying infrastructure play a central role since the contexts where they perform their functions are hyper-connected. There is a need to understand their interaction, patterns, constraints and characteristics so better systems are deployed, enhancing its reliability and performance throughout their environment. The main application of PE is for *dimensioning* systems or *Capacity Planning* [4, 5], estimating resources as demand increases (e.g. work that must be dealt with, arriving for processing at stations).

The aim here is to provide a set of introductory notions of fundamental concepts behind PE applicable to the quantitative analysis of general-purpose systems. Another issue of concern is that (more or less) the same definitions are being *resurrected* with novel trendy and modern nomenclature for addressing already established concepts, e.g., *cyber-resilience* and *cyber-survivability* (*rant:* everything nowadays needs to have *cyber* in the name), system *hardening*, and so on. *Responsiveness* is another attribute that has different meaning depending on the audience. For user experience (e.g. *Graphical User Interface* – GUI) and application developers (apps), it means that the interface adapts to different screen sizes (TV, tablets, cell phones) seamlessly whereas for performance testing it means to finish a task in due time.

The majority of PE literature poses challenges to newcomers as they find it hard to follow key concepts and apply those in real world problems. Figure 1 illustrates this notion, abstracted into learning how to draw an owl. Authors choose a pace to convey ideas that it does not resonate with non-experienced audiences, e.g., it is too fast to follow (I hope present work will not follow the same path though).

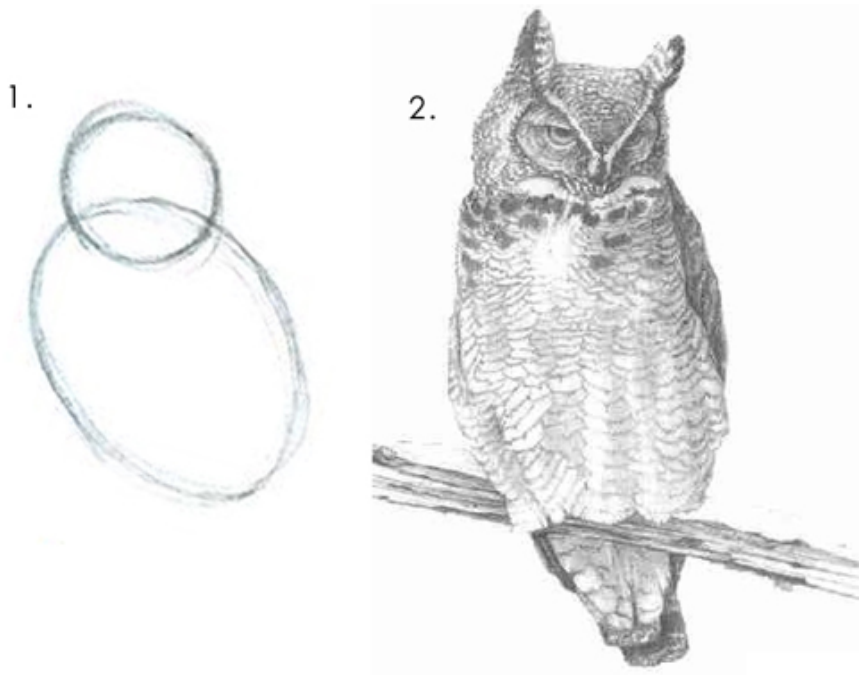

Figure 1: Learning to draw an owl comparing to learning PE (source: unknown).

The idea here is to explain in detail the basic notions behind PE for newcomers and early practitioners, demonstrating its strengths with examples. Our aim is to discuss Markovian based approaches and discrete event simulation in a practical manner.

This work is divided as follows: It starts in Section 2 with a discussion of *Non-Functional Properties* or attributes of systems. Then, common approaches to *Performance Evaluation* are presented on Section 3. A discussion on *Markov Chains* (discrete and continuous) is detailed on Section 4 whereas Section 5 explains *Discrete Event Simulation*. The work is finalised in Section 6 with concluding remarks.

1.1 Concepts and application domain

According to ISO24765:2010's glossary [2], a differentiation is required for *error*, *fault* and *failure*.

An ***error*** is *1. a human action that produces an incorrect result, such as software containing a fault. 2. an incorrect step, process, or data definition. 3. an incorrect result. 4. the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition*.

A ***fault*** is *1. a manifestation of an error in software. 2. an incorrect step, process, or data definition in a computer program. 3. a defect in a hardware device or component. Synonym: bug*.

A ***failure*** is *1. termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. 2. an event in which a system or system component does not perform a required function within specified limits (observation: a failure may be produced when a fault is encountered - see also [6])*.

Although PE could address virtually any system, one common interest is devoted towards *computational systems* and *software*, which is the focus of the present work. Before starting the concept of *Validation, Verification and Accreditation* (VV&A) must be discussed [1]. *Validation* concerns preoccupations as to the realisation of what is desired by customers, clients, users, i.e., *what was requested is consisting with what was done? Verification* aims to discover whether or not what was built or implemented does fulfil functional requirements (*what was built reflects what was contracted?*). *Accreditation* is about belief in the functional characteristics of systems, i.e., *does the implemented system is credible and trustworthy according to an authority or organisation?*

One could inspect a system for its functional requirements or its non-functional properties (more information on Section 2). In terms of its *functions*, the objective of software analysis is to validate and verify systems, attesting to its present functionalities and discovering *faults* (defects, bugs) which diminishes its quality and cause money losses.

There are two basic techniques to analyse software systems, dynamic or static. In the *dynamic* approach, the system is executed and *Software Testing* techniques (functional testing or unit testing, respectively black-box and white-box approaches[1]) are used to test input/output to a small set of available functionalities offered by the system. In the static approach, the system is not executed, however, a specification is devised and then subjected to *Formal Methods* techniques, for example, *Model Checking* [7] or *Theorem Proofing*. The idea is to exhaustively test *all possible entries* and search for non-obvious inconsistencies, deadlocks and errors (if not all, at least a statistically relevant state subset using *Statistical Model Checking* [8] techniques).

Formal Methods are related to the study of *Discrete Event Dynamic Systems* (DEDS) and *Formal Verification of Systems*. Examples of DEDS are Automata theory, Supervisory Control Theory, Petri nets, Discrete Event System Specification, Markov Chains, Queueing theory, Discrete Event Simulation and Concurrent Systems. Formal Verification uses Finite State Machines (Moore, Mealy), Model Checking, Linear Temporal Logic (LTL) or Computational Tree Logic (CTL).

The focus here is devoted towards DEDS and performance evaluation approaches.

2 Non-Functional Properties of systems

A *Non-Functional Property* (NFP) or *attribute* refers to a quality[2] aspect of a system instead of its functionality, which, for computing, is tackled by software implementation. There is a plethora of NFP of systems and examples are those described in *dependability* research [9], for instance *reliability*, *integrity*, *availability*, *maintainability* and *safety* of systems (not ordered by importance). Other worth noticing properties are *performance*, *usability* and *security* (which is aimed at investigating issues arising in *confidentiality, integrity and availability* – known as the CIA triad – plus *non-repudiation*).

Integrity is the absence of improper system alterations (on purpose or unwillingly). *Availability* is readiness for correct service. *Confidentiality* is absence of unauthorized disclosure of information (or access to data). *Non-repudiation* is the association of action to unique individuals. *Reliability* is continuity for correct service. *Maintainability* is the ability to undergo repairs or modifications. *Safety* is absence of catastrophic consequences to users. *Performance* aims to quantify the amount of useful work yielded by resources, how long it took it and how well did it perform. *Fault-tolerance* aims to avoid service failures in the presence of faults. *Resilience* is a synonym to fault-tolerance whereas *robustness* is a close definition, directed at the ability of a system to cope with errors (see definition above) during execution. Other interesting NFPs are *scalability*, *adaptability*, *portability*, *survivability*, *efficiency* and *sustainability*, *modularity*, *flexibility*, *portability* (*compatibility*), *recovery*, *resistance*, *accessibility*, to list a few.

Figure 2 shows some relevant quality properties of systems. Notice that dependability overlaps availability and integrity with security. It is usual to investigate dependability plus security, fault-tolerance and performance concerns in computing systems. The major *threats* that cause deterioration of system dependability are *errors*, *faults* and *failures* (defined in Section 1.1) [9].

In summary, managers, stakeholders, analysts, modellers, users, customers would like to use systems presenting *continuous, secure, safe and reliable timely services, equipped with user-friendly interfaces for efficiency, efficacy*[3] *and productivity (among other objectives).*

Other systems types are present in the literature addressing similar characteristics. For example, much is said about *critical* systems, mentioned in many works. These systems invoke the notion of *safety* because their incorrect functioning affect human (or other life form) existence. *Mission critical* systems are essential systems that have profound impact (financial or vital) if they cease operating. Finally, so called *complex* systems are systems with non-obvious multiple transitions, usually defined in massive sizes.

To that effect, *what is a system?* A *system* is a set of interacting components that collaborates to perform a function that addresses a problem. It can be abstracted or simplified into a *model*. When *abstracting*, one focus on a *high-level* perspective of the *System Under Study*, not taking into account unimportant details. Famous painter Pablo Picasso produced an interesting study on abstraction, as seen in Figure 3. In the work called *The Bull*, from 1945, he presents a series of lithographs (works produced using paint or oil on stone or metal canvases). It is interesting to see that all representations still show remarkable resemblance to a bull, in different levels of detailing used to draw each one.

[1] *Black-box* approaches test the system without inspecting the source code whereas *white-box* not only observes the code but also inserts new testing specific functions/methods to capture *defects*.

[2] The actual definition of *quality* has been discussed throughout human history, in philosophy and other fields.

[3] *Efficiency* is considered the optimal transformation of input into outputs, whereas *efficacy* is the ability to reach successful results despite costs.

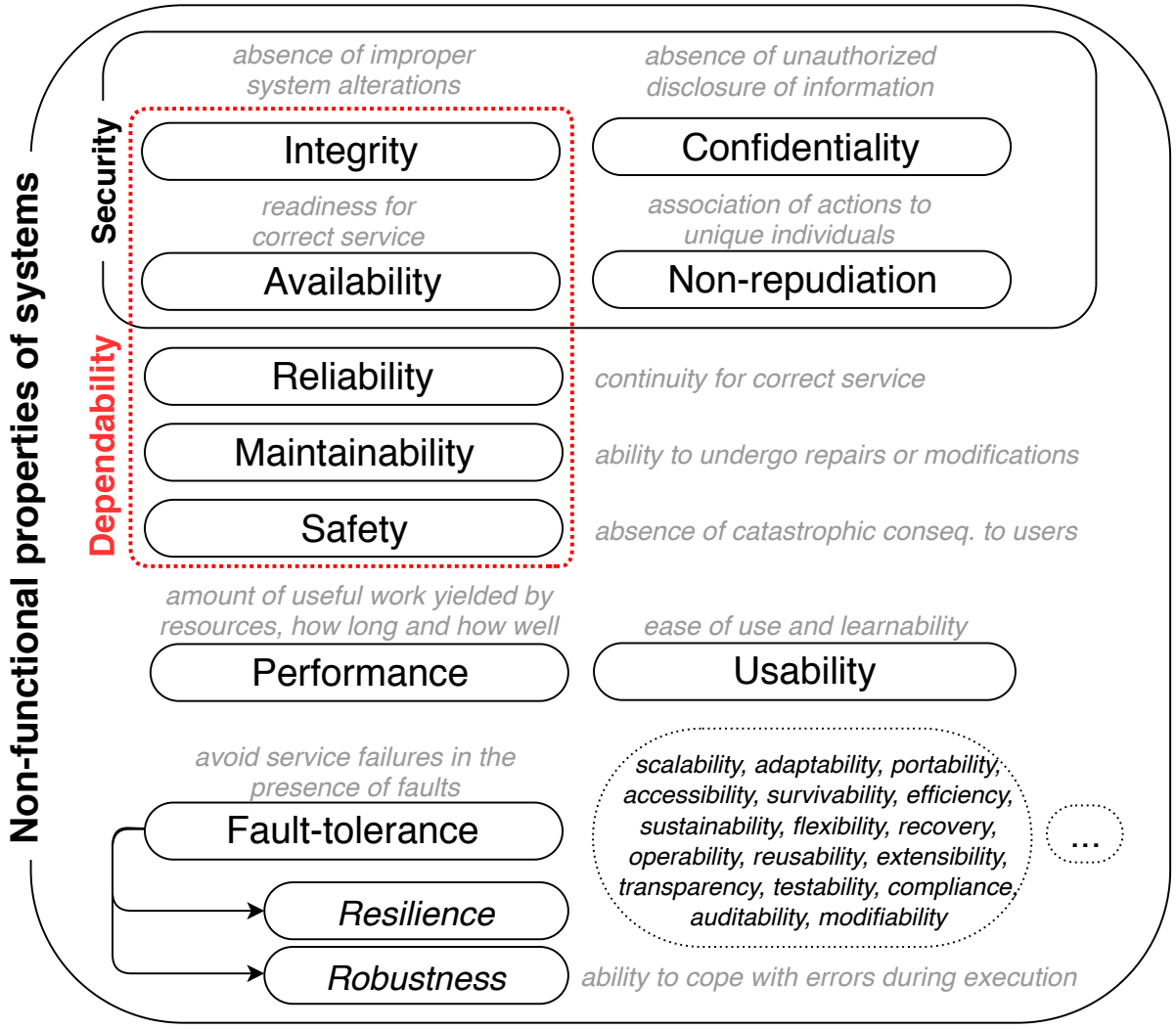

Figure 2: Selected quality properties of systems (based on [9]).

Abstraction is closer to *art* than to *science*, where seeing less is usually more important than detailing subjects. Modelling systems could be seen as an art, because one must select fundamental operations to be under the scope.

The *level-of-detail* (LoD) is chosen by the modeller, if too small, it could be hard to evaluate its characteristics, if too coarse, it doesn't capture its functionalities. So, it is the onus of the analyst to select the most adequate version. System *decomposition* consists on break it into simpler and more manageable parts or sub-systems (or modules, components, and so on). The act of *modelling* consists on capturing a system's *operational semantics* or *behaviour* and translate it to *primitives*. The *behaviour* of a system is the set of steps it must perform to implement its *function*, normally described by a set of states and transitions[4]. Transitions are related to changes a system evolve from state to state. The *state space* is the combination of all sub-system states a system may assume. Many problems have massive state spaces yielding *state space explosion* problems.

Modelling has two phases, *construction*, where one creates the stochastic processes that govern its behaviour, and processing, where system *measures* of interest are obtained. *Metrics* or *measures* could be many attributes usually throughput, utilisation, response time, average population or a customised characteristic. In close relation to this, there are so called *Key Performance Indicators* (KPIs) one could use to derive *indices* in many heterogeneous situations.

[4]In graphs terminology, *states* are *vertices* and *transitions* are *directed edges* whereas in communication networks, participating entities are *nodes* connected by *links* – e.g. the abstraction is more or less the same.

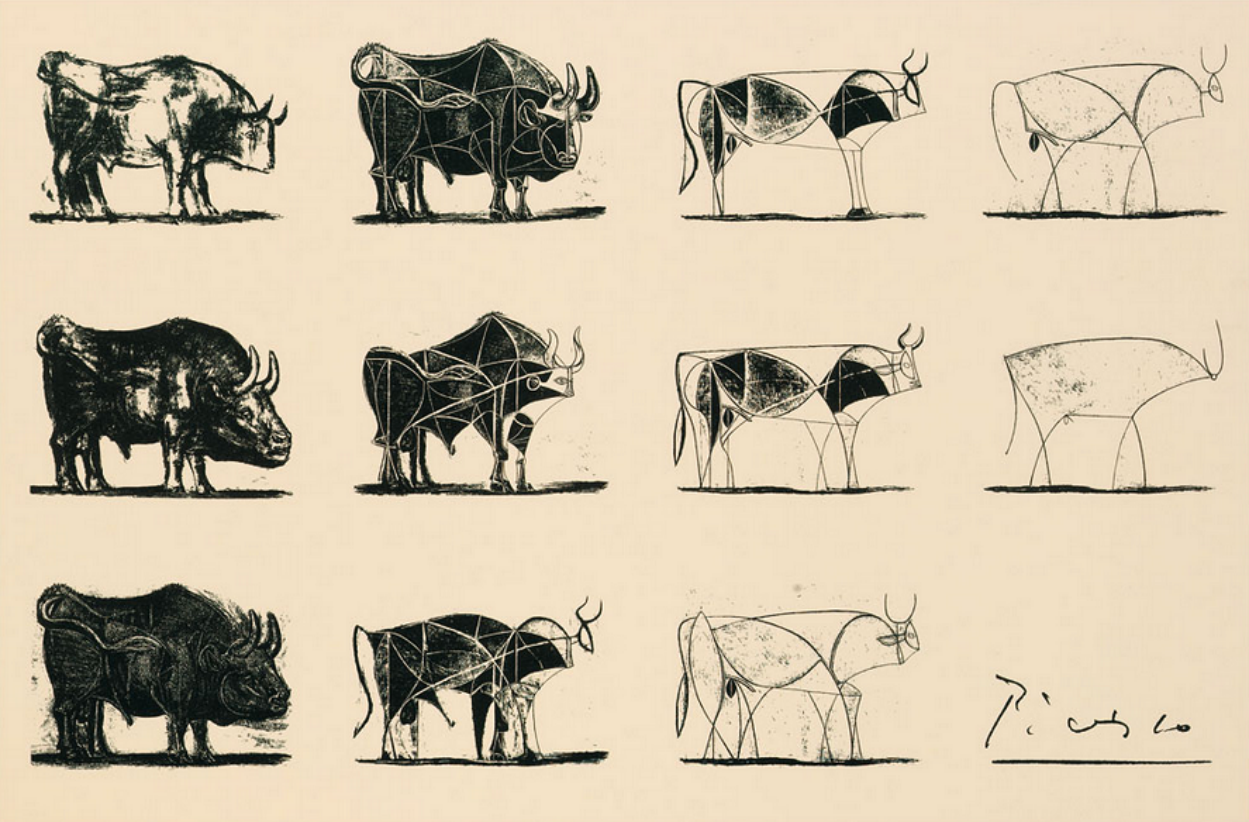

Figure 3: Various stages of a bull, according to Pablo Picasso.

For example, KPIs in healthcare could be a patient's *Length of Stay* (LoS) or *Waiting Time* (WT), the percentage of admissions leading to death, or the average cost in *Accidents & Emergency* (A&E) departments.

The first step when evaluating a system is to choose how to represent concepts and behaviour. So, one choose a modelling *formalism* to depict semantic to states and transitions. Researchers employ these sets of rules to address various problems, from economy to information retrieval [10] (i.e. Google's PageRank algorithm) among many other. Examples of formalisms that employ *state-transition* primitives are *Markov Chains* (MC), *Birth-Death Processes* (BDP), *Queueing networks* (QN) or *Petri nets* (PN). Their stochastic versions differ from *Labelled Transition Systems* (LTS) due to the assignment of numerical values (e.g. rates governed by probabilistic distributions or probabilities) in transitions instead of symbolic texts. In an LTS (also known as *Kripke structure* or *Finite Automata*) the labels on transitions allow users to model non-determinism, so they are a powerful formalism employed in system verification and formal analysis due to its scalability and applicability properties [11].

The evaluation pipeline starts with: 1) formalism definition followed by 2) modelling behaviour, 3) assignment of parameters (assuming data was already being processed, cleaned and treated), 4) numerical execution, 5) interpretation and 6) output analysis. If required or if results do not make sense or are somehow invalid, it is usual to maybe change the formalism, review the model, assign other parameters, choose another numerical method or perform model refinements to capture other behaviour.

One interesting aspect of performance evaluation process is the ability to conduct "what-if" analysis by the definition of multiple scenarios through parameter (or states or even transitions) variations. Then, with the results, the modellers could investigate better or worse configurations and present their findings to decision makers or other stakeholders. In a nutshell, PE equips analysts with quantitative measures that largely influence decisions across many domains.

3 Approaches to performance evaluation

The literature details approaches for performance evaluation or performance appraisal [3]. There are three PE techniques: monitoring, analytical modelling and simulation. *Emulation* could be used, which is directed towards faithful *imitation*

of a system, however, despite its significance, it is out of the scope of this work to discuss it. In a broad scope, when evaluating systems, one could perform experimentations directly or through modelling, as explained below.

- Quantitative Evaluation of Systems
  - Direct experimentations
    - **Monitoring**
  - Experimentation through modelling
    - Physical model (e.g. a scale model or a mock-up model)
    - Mathematical model
      - **Analytical model**
      - **Simulation**

The first question one should ask is *Why embark in a performance evaluation investigation?* The main answer could be to use it to test designs before physical implementation or costly resource acquisition or hiring. Another interesting objective could be to devise alternative scenarios comparing them with baseline models and compute indices that numerically corroborate or refutes better configurations or modifications. Also, one could study where key resources are better used or situations where costs are reduced just by changing the positioning of a given machine in the floor plant of an industry. Managers do not want to incur in expensive endeavours only to discover that the new proposition is in fact worse than the previous one (at production). In this sense, PE is an important approach aimed at better decision making applicable for a wide range of situations. In terms of actionable techniques, the list below describes the major ones related to PE of systems.

1. Monitoring and instrumentation [12]
   - Physically
     - *Sensors* could be employed to sense environment and monitor a system, e.g., a building.
     - *Other devices*, e.g., turnstiles to constrain access, walls and checkpoints, and so on.
   - Logically (e.g. in software)
     - *Performance Counters* (e.g. *Disk Queue Length*, process *Context Switches*, and so on).
     - *Profiling* where additional code is automatically inserted into a software to yield performance measures during execution (e.g. system calls, percentage of time spent in function/method, and so on).
     - *Customised* approaches inserting control directives (e.g. *prints* in specific places within a software).
2. Analytical Modelling (Markovian based approaches)
   - Markov Chains
     - *Birth-Death Processes*, *Queueing Networks*
     - *Continuous Time Markov Chains* (CTMC), *Discrete Time Markov Chains* (DTMC)
   - *Structured* approaches (generating and working with an underlying MC)
     - *Stochastic Petri Nets*[5] (SPN) [13], *Stochastic Automata Networks* (SAN) [14, 15], stochastic process algebras (e.g. *PEPA* [16]) as well as generalised approaches such as *Reactive Modules* (RM) [17].
3. Simulation
   - *Discrete Event Simulation* (DES), *Discrete Time Simulation* (DTS), *Monte Carlo Simulation* (MCS), *Agent Based Simulation* (ABS), *System Dynamics* (SD), *Dynamic Systems* (DS) or a combination of techniques (*co-simulation* frameworks for example MECSYCO [18], INTO-CPS [19] or DEVS [20]).

There are several trade-offs when considering which approach is better suited for a given study. For example, depending on the problem, monitoring and instrumentation could be hard to be deployed as well as expensive (assuming it is even possible to reach key components, e.g. inside a nuclear power plant). It yields very good results if the extra instrumentation does not affect performance (known as the *paradox of monitoring*).

3.1 Discussion

Analytical modelling is known for providing low LoD to modellers (i.e high abstraction), however, being a mathematical representation of a system, it usually produces results in a timely manner (in contrast to the other two approaches).

[5] It is *stochastic* (e.g. a synonym of random) because the firing rules for transitions are governed by probabilities distributions based on the assigned rates.

Depending on the formalism, product form equations are used to derive pertinent indices very quickly. If closed equations (e.g. *product form*) are not present in the formalism, then modellers must resort to numerical solvers. They are executed according to a set of parameters, such as difference between iterations or maximum number of iterations. If the solver finishes, it produces the steady state of the system, i.e., it converged, reaching the stationary regime. This means that it yielded the system's *occurrence probabilities*, i.e., the system does not distinguishes the initial state it used for the solver execution and it has ran steadily after a very long run (e.g. infinite). On the other hand, if it did not finish execution even after the parameters were adjusted (e.g. increasing the number of iterations or reducing the observed difference between iterations), it has *diverged* and another mechanism such as *transient analysis* must be employed.

Finally, simulation has some advantages and also drawbacks. For example, due to use of programming, several defects are potentially introduced into the evaluation, causing difficulties for validation. And due to its highly software intensive execution nature, it is very computationally expensive. Besides all that, after execution, one must also run descriptive statistical analysis techniques to produce confidence intervals, averages, standard deviations, and so on.

Modellers should balance trade-offs when choosing the PE technique best suitable for the problem at hand. In terms of validation, ideally, one technique is used to validate the other, e.g., simulation is used to confirm results obtained from the analytical model and so forth. Such broad investigations take a large amount of time to be processed. However, they must be done to ensure that model refinements and modifications will produce valid output and thus better decision making.

4 Markov Chains

The formalism of *Markov Chains* (MC) was conceived by Andrey Andreyevich Markov (1856–1922), in a publication addressed to the Imperial Academy of Sciences in St. Petersburg (Russia) on 23^{rd} of January, 1913. Markov originally applied the idea to Alexander Pushkin's novel *Eugene Onegin* (written originally in the Russian language)[6]. Markov himself spent hours analysing consonants and vowels within the book (≈ 300 pages) to devise its matrix and calculate probabilities of their occurrence.

MC is a powerful modelling formalism to address behaviour of systems with simple primitives [21, 22, 23, 24]. MC is known since early 20^{th} century for multiple purposes. It was applied in *time shared systems* by mid 1960, in the *Massachusetts Institute of Technology* (MIT), for scalability analysis [25, 26]. Solution are applied to MCs with special characteristics, such as not having *absorbing* states for example. Also, the MC must be *ergodic*, i.e., *irreducible*, with *positive recurrent* and *aperiodic* states (there is an extensive literature detailing these properties).

4.1 Markov property and the exponential distribution

The notion behind the Markov property it is the fact that the past is irrelevant to determine the future evolution of a system, as stated earlier. It is central in Markov modelling with repercussions on the obtained solution and assumptions one conduct in an analysis effort. The Markov property is modelled using the exponential distribution because it is memoryless (there is a wealth of literature explaining and detailing these notions in a reasonable level of detail [27, 28]).

An example: a commuter takes a bus going to work that has a non-deterministic behaviour as to its inter-arrival times at the stop. The bus would pass every 20 min or so, however, it has never respected this pattern deterministically. When the commuter have arrived at the bus stop and looked at its schedule, and it said that it would come in the next 5 min, he simply knew that it could take 1 min, 10 min, or even 45 min (to his dismay). One could model this situation using exponentially distribution parameter for this time (adjusted to its rate). The average value for this parameter could be equal to 20, and then one could compute the probability that the bus would arrive in the next 5 min, 10 min, or any other value (of course, the probability of arriving less than 3 hours should be near 100%).

In probability there are two notions of particular interest to distributions: the Probability Density Function (PDF) and the Cumulative Distribution Function (CDF). In a nutshell, the PDF expresses the likelihood of a random variable being drawn from a distribution whereas the CDF is the probability that it will take up to a value for something to occur. The data on Table 1 was built on MS-Excel using the function `=EXPON.DIST(x,`λ`,CUMULATIVE)` where one selects the wanted parameter (x), the value of X (P(X<x)) and whether the CDF (`CUMULATIVE=true`) of PDF (otherwise) will be computed.

Observe how close the PDF is from the parameter ($T = 20$, $\lambda = \frac{1}{20} = 0.05$) whereas the chance of the event happening (CDF) in let's say, less than 10 min (P($X \leq 10$) is 0.3934 (39.34%). And, the probability of it happening between 15 min and 25 min equals to P($X \leq 25$) - P($X \leq 15$) = $0.7134 - 0.5276 = 0.1859$ (18.59%). There is a higher chance of the bus taking up to 30 min (77%) to arrive – the limiting value approximates to 1 (100%) – which captures real settings

[6] An interesting description in `https://www.americanscientist.org/article/first-links-in-the-markov-chain`.

Table 1: Exponential distribution (CDF and PDF) for rate parameter of $\lambda = 0.05$ (T=20), for modelling the previous bus example – values are rounded to 4 digits.

x	CDF P(X<x)	PDF		x	CDF P(X<x)	PDF		x	CDF P(X<x)	PDF
1	0.0487	0.0498		11	0.4230	0.0489		21	0.6500	0.0484
2	0.0951	0.0497		12	0.4511	0.0488		22	0.6671	0.0483
3	0.1392	0.0496		13	0.4779	0.0488		23	0.6833	0.0483
4	0.1812	0.0495		14	0.5034	0.0487		24	0.6988	0.0482
5	0.2211	0.0494		15	0.5276	0.0486		25	0.7134	0.0482
6	0.2591	0.0493		16	0.5506	0.0486		26	0.7274	0.0482
7	0.2953	0.0492		17	0.5725	0.0485		27	0.7407	0.0481
8	0.3296	0.0491		18	0.5934	0.0485		28	0.7534	0.0481
9	0.3623	0.0491		19	0.6132	0.0484		29	0.7654	0.0481
10	0.3934	0.0490		20	0.6321	0.0484		30	0.7768	0.0480

nicely. Finally, the probability of taking less than 3 hours is 0.9998 (99.98%), i.e., it will certainly take less than this to arrive.

It was shown the CDF and the PDF for the exponential distribution, however, how one draw values from it? It uses the uniform distribution and the natural log of a number to draw values from the distribution, where it averages to the chosen parameter β. For example, the following formula generates numbers from the exponential distribution: $F = -\beta \times ln(1 - U(0; 1))$ where `ln` is the log number base e and U is a function that returns random numbers from zero to one from the uniform distribution.

Table 2: A set of 100 values drawn from the exponential distribution with parameter $\beta = 20$

16.0921	3.8792	0.5045	5.2710	3.4098	45.2259	22.3523	9.7070	53.0025	8.9746
6.5102	3.9326	1.7833	35.1001	13.3477	27.5008	13.8040	5.6191	12.7323	48.7543
52.7689	35.6958	46.4578	29.6118	26.6120	21.0565	8.7866	1.1200	12.3297	18.5835
56.3307	3.3483	0.4288	8.6211	16.7254	2.6362	4.5913	78.2304	88.6404	29.5169
10.9735	17.3608	22.0088	3.7012	33.3508	16.7844	7.6594	29.3216	9.0739	12.0181
0.9138	4.2864	16.7315	47.2680	9.2295	7.8907	0.4522	92.7094	22.1309	6.3369
0.2632	4.5856	21.5330	24.4209	6.7903	56.9635	27.2254	54.7721	20.9724	18.4612
5.7797	1.5574	26.5981	12.7389	26.2620	38.6494	4.8212	7.2107	15.6482	1.5422
34.9726	3.7010	24.7590	14.5029	27.6289	10.7510	20.7245	29.5924	18.1422	3.7090
1.2829	60.5759	21.7355	11.4271	21.7182	34.2270	0.8822	48.9102	2.4312	9.8077

Table 2 shows 100 draws from the exponential distribution with parameter $\beta = 20$ (these values average to 20.2807). So, back to the example, one could consider those values drawn from the distributions to act as the time it took for the bus to arrive. The formula in MS-Excel for doing this is `=-B*LN(1-RAND())` (where B is the cell with parameter β).

4.2 Discrete Time Markov Chains

Discrete Time Markov Chains (DTMC) uses probabilities to decorate transitions symbolising state change in order to model system behaviour. Each line of the model must sum to one, if not, the row values should be uniformised before any solution technique is employed. The idea is to define a *stochastic matrix* D and subject it to solution, which could be powering it to the n-th power, multiplying iteratively by a vector or using simulation. All these three solution methods are inspected as follows, commencing with the power method.

The idea of the power method is to multiply the stochastic matrix D by itself to the n^{th} power, i.e., D^n, until the process starts repeating itself. Hopefully, we will be able to achieve convergence and use results for system *reasoning*.

Let's suppose D to be any matrix where the sum of each of its rows equals to one, for example[7]. It represents a three state problem (namely 'A', 'B' or 'C') having the following routing probabilities (i.e. [A,B,C] x [A,B,C]):

$$D = \begin{bmatrix} 0.470588235 & 0.176470588 & 0.352941176 \\ 0.470588235 & 0.0 & 0.529411765 \\ 0.176470588 & 0.117647059 & 0.705882353 \end{bmatrix}$$

[7]This is just an example of a model to represent a DTMC, for its CTMC, refer to Section 4.3.

So, computing $D^2 = D \times D$ (and so forth), yields

$$D^2 = \begin{bmatrix} 0.366782007 & 0.124567474 & 0.508650519 \\ 0.314878893 & 0.14532872 & 0.539792388 \\ 0.262975779 & 0.114186851 & 0.62283737 \end{bmatrix}$$

$$D^3 = \begin{bmatrix} 0.307515475 & 0.121873541 & 0.570610984 \\ 0.303205182 & 0.121981298 & 0.57481352 \\ 0.296200955 & 0.120472695 & 0.583326349 \end{bmatrix}$$

$$D^4 = \begin{bmatrix} 0.300533975 & 0.121087336 & 0.578378689 \\ 0.300485961 & 0.12108146 & 0.578432579 \\ 0.300396145 & 0.121069373 & 0.578534482 \end{bmatrix}$$

$$D^5 = \begin{bmatrix} 0.300448443 & 0.121076235 & 0.578475322 \\ 0.300448436 & 0.121076234 & 0.57847533 \\ 0.300448423 & 0.121076232 & 0.578475345 \end{bmatrix}$$

$$D^6 = \begin{bmatrix} 0.300448443 & 0.121076235 & 0.578475322 \\ 0.300448443 & 0.121076235 & 0.578475322 \\ 0.300448443 & 0.121076235 & 0.578475322 \end{bmatrix}$$

In six steps (powering the matrix D to the 6^{th} power), the rows stabilised, i.e., they have fixed values, without observable variations among them, so the solution has *converged*. It will not matter to continue the process of powering this matrix because the values will not change (this is the *stationary regime*). This matrix corresponds to the so-called *eigenvector* of Q, i.e., the vector that corresponds to the matrix (in unsophisticated terms).

The interpretation of the results obtained is: the chance of the system being at any time at the first state ('A') is 30.04%, in 'B' is 12.10% and in 'C' is 57.84%. These values were not obvious when we modelled the problem using rates, i.e., we could not compute the permanence probabilities by just inspecting the state frequencies.

It is worth mentioning that this is just one way of producing results. For a DTMC, one could compute $\pi D = \pi$ for an initial vector π having equiprobable values or just having it summing to one. Next, it is demonstrated how to use simpler data structures to solve this problem and achieve the same results.

This approach is called *Vector-Matrix Product* (VMP), and consists on using an initial vector (having values that sum to one) multiplied by a stochastic matrix. The advantage of such technique is being able to address the same problem in a computationally less complex operation since only one matrix row is used instead of all matrix. So, the idea is to use a vector V and multiply it iteratively by the stochastic matrix D, i.e. $\pi D = \pi$ or in this case, $V \times D = V$:

$$V \times D = \begin{bmatrix} 0.0 & 0.0 & 1.0 \end{bmatrix} \times \begin{bmatrix} 0.470588235 & 0.176470588 & 0.352941176 \\ 0.470588235 & 0.0 & 0.529411765 \\ 0.176470588 & 0.117647059 & 0.705882353 \end{bmatrix}$$

Now, one performs the computation of the vector V by matrix D until the vector values between iterations are the same (convergence). For achieving results a simple numeric spreadsheet is sufficient to compute the VMP:

1	0.0000000	0.0000000	1.0000000
2	0.1764706	0.1176471	0.7058824
3	0.2629758	0.1141869	0.6228374
4	0.2874008	0.1196825	0.5929168
5	0.2962010	0.1204727	0.5833263
6	0.2990217	0.1208974	0.5800810
7	0.2999762	0.1210133	0.5790105
8	0.3002910	0.1210559	0.5786531
9	0.3003961	0.1210694	0.5785345
10	0.3004310	0.1210740	0.5784950
11	0.3004426	0.1210755	0.5784819
12	0.3004465	0.1210760	0.5784775
13	0.3004478	0.1210761	0.5784761
14	0.3004482	0.1210762	0.5784756
15	0.3004484	0.1210762	0.5784754
16	0.3004484	0.1210762	0.5784754

After 16 iterations, the vector converges to the recurrence probabilities (observe that for the residual difference between iterations 15 and 16 is negligible)[8]. This method takes longer to converge, however, its complexity is inferior to that of matrix power specially for high order problems.

For the initial vector used in the example it was necessary a total of 16 steps of a VMP. However, notice that if another initial vector was chosen, the results could take less iterations to complete. So, if one uses *preconditioning techniques* for initialising the vector, one could accelerate the process and reach results earlier.

It is possible to implement a software to do this process according to a residual difference between iterations. The code for performing the VMP is shown next written in *The C Programming Language*.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>     // memcpy
#include <math.h>       // pow, sqrt

#define ORDER 3         // model size
#define MAXRUNS 1000000 // number of runs
#define RESIDUE 1e-10   // residual difference between two iterations

int converge(float *r1, float *r2) {
   int i;
   for (i = 0; i < ORDER; i++)
      if (sqrt(pow(r1[i] - r2[i], 2)) > RESIDUE)
         return 0;
   return 1;
}

void multiply(float *v, float m[ORDER][ORDER]) {
   int i,j;
   float aux[ORDER];
   for (i = 0; i < ORDER; i++) {
      aux[i] = 0;
      for (j = 0; j < ORDER; j++)
         aux[i] += m[j][i] * v[j];
   }
   memcpy(v, aux, sizeof(float) * ORDER);
}

int main(int argc, char *argv[]) {
   float D[ORDER][ORDER] = {
      { 0.470588235, 0.176470588, 0.352941176 },
      { 0.470588235, 0.0        , 0.529411765 },
      { 0.176470588, 0.117647059, 0.705882353 }
   };
   int i;
   int runs = 0;
   float pvec[ORDER];
   float old[ORDER];
   pvec[0] = 1.0;  // initialise first position
   for (i = 1; i < ORDER; i++)
      pvec[i] = 0.0;
   do {
      memcpy(old, pvec, sizeof(float) * ORDER); // copy to old array
      multiply(pvec, D);
      if (++runs > MAXRUNS)
         break;
   } while (!converge(old, pvec));
   printf("Number of iterations: %d\n", runs-1);
```

[8] As a matter of fact, it takes 21 iterations to reach convergence to $1e^{-10}$, not shown here due to space restrictions.

```
   for (i = 0; i < ORDER; i++)
      printf("%d=%f\n", i, pvec[i]);
}
```

Function `convert` takes two arrays (previous and current) and tests whether or not the difference between them are below a threshold (set by the constant RESIDUE), which in this case is set to $1e^{-10}$. Function `multiply` takes an array as parameter and multiplies it by the stochastic matrix. This process is repeated until `MAXRUNS` is reached or the difference is less than $1e^{-10}$, whichever condition is reached first.

Finally, it is shown a simulator in C that uses matrix D as input and a random walker starting from an initial state (in this case is 0) uses the probabilities to randomly decide where to go next and saves each visit choice in an array.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define ORDER 3         // model size
#define MAXRUNS 1000000 // number of runs

int main(int argc, char *argv[]) {
   float D[ORDER][ORDER] = {
      { 0.470588235, 0.176470588, 0.352941176 },
      { 0.470588235, 0.0        , 0.529411765 },
      { 0.176470588, 0.117647059, 0.705882353 }
   };
   int i;
   float r;             // variable for saving random number
   int runs = 0;
   int state = 0; // starts at 0 state (could be any -- selected at random)
   int visits[ORDER];
   float acc;
   srand(time(NULL));          // set initial random seed -- based on now
   for (i = 0; i < ORDER; i++)   // states initialisation
      visits[i] = 0;
   while (runs++ < MAXRUNS) {
      r = (float) rand()/RAND_MAX;
      acc = 0.0;
      for (i = 0; i < ORDER; i++) {
         acc += D[state][i];
         if (r < acc) {
            visits[i]++;
            state = i;
            break;
         }
      }
   }
   for (i = 0; i < ORDER; i++)
      printf("%d=%f\n", i, (float)visits[i]/runs);
}
```

After compilation (`gcc -o randomwalk randomwalk.c -O3`), the process yields the following results:

```
0=0.300485 // for 'A' state, i.e. 30.04%
1=0.120965 // 'B' state or 12.09%
2=0.578524 // 'C' state, with 57.85% (this is the most visited state)
```

Note that the results are close to the ones obtained through the power method earlier for 1000000 runs (it would be even closer if one selected a higher number of runs). It is possible to use this simple code to run any DTMC, it suffices to change the variable D and the constant MAXRUNS and it would produce permanence probabilities. It is recommended to run the process for large MAXRUNS to determine whether or not the results have stabilised numerically.

4.3 Continuous Time Markov Chains

Not always it is suitable to work with probabilities to address system behaviour. Sometimes, one is interested on the time spent on each state or on the rate (frequency) of transiting from state to state. For those cases, *Continuous Time Markov Chains* (CTMC) are used. The idea is similar to DTMC where the starting point are the states and transitions, however, for CTMC, one assign numerical values to the transitions also called rates. Next, a rate matrix is defined and then modified to become an *Infinitesimal Generator* (IG), i.e., a special matrix where each line sums to zero. In order to reach that property, one should adjust every line by inserting on the matrix's diagonal the sum of the line multiplied by -1. With the IG, one must search for the probability vector and there are several ways of doing it. For example, one could devise a set of linear equations and solve it using a mathematical library (e.g. Matlab, Mathematica, Maple, GNU/Octave), which will employ a solution method (GMRES, SOR, Arnoldi, Gauss, etc). One could translate the IG to its *Embedded MC*, yielding the DTMC representation of the CTMC and then applying the methods discussed in Section 4.2.

More formally, a CTMC is a stochastic process equipped with the *Markov property* employing a probability distribution with the *memoryless property* such as the exponential distribution [22]. This property is defined as:

$$P[X(t_k) = s_k \mid X(t_{k-1}) = s_{k-1},\ \ldots\ , X(t_0) = s_0] = P[X(t_k) = s_k \mid X(t_{k-1}) = s_{k-1}]$$

where $\{\ X(t) \mid t \in \mathbb{R}_{\geqslant 0}\ \}$ are a set of random variables, $X(t)$ are observations at time t (where $X(t)$ is the state of the system at time t). The meaning of the memoryless property is that, assuming a random user set to land on a given state, the decision of visiting the next state does not take into account the visited states, meaning that the past is irrelevant up to this point. A CTMC decorates its transitions with exponentially distributed (which is a memoryless probability distribution) delays or rates in place of labels (a characteristic particular to LTS).

A CTMC is equipped with a set of discretely defined enumerable states where time evolves in a continuous fashion according to the assigned set of rates [29]. Formally, a CTMC consists of a tuple (S, s_0, R, L) where

- S is the state space of the problem corresponding to the finite set of states present in the model,
- $s_0 \in S$ is defined as the initial state,
- $R : S \times S \to \mathbb{R}_{\geqslant 0}$ maps the transition rate matrix of the model, and
- $L : S \to 2^P$ are labels mapping states to atomic propositions in P.

If multiple states have ongoing transitions with different rates, i.e. $R(s, s') > 0$, then a *race condition* occurs, where the transition with the least value (according to the exponential distribution of the rate) is triggered first. The time spent in a state is drawn from the exponential distribution given by $E(s) = \sum_{s'} R(s, s')$, where $E(s)$ is the exit rate of the state s and the probability of leaving a state s within $[0, t]$ equals to $1 - e^{-E(s)t}$

A CTMC encloses a discrete time counterpart termed the *Embedded DTMC*, computed using the transition rate matrix elements and the maximum value (negated), i.e., dividing each cell element by such value. The so called *Infinitesimal Generator* $\mathbf{Q}$ of a CTMC is a matrix $S \times S$ whose non-diagonal entries are the same as in R, and the diagonal entries are defined in such a way that the sum of each row of $\mathbf{Q}$ equals 0.

$$\mathbf{Q}(s, s') = \begin{cases} R(s, s') & s \neq s' \\ -\sum_{s \neq s'} R(s, s') & \text{otherwise} \end{cases}$$

These equations are related to the operational meaning of CTMCs. In order to solve $\mathbf{Q}$ and compute results, one must multiply it by a vector containing at least one non-zeroed entry (or equiprobable, i.e., all values are the same) termed π, calculating $\pi\mathbf{Q} = 0$ iteratively until a balance is reached (the difference between each iteration reaches a negligible residue ϵ defined by the modeller[9]).

The stationary result (also called steady state) in CTMC yields the *occurrence or permanence probabilities* for the set of modelled states. So, in the end of the process, if convergence was reached, those results represent the system's behaviour in *infinite*, e.g., after a long execution. Otherwise the model diverged, and the modeller should revise the rates or states of the model or drive transient analysis.

[9] Assuming S as the state space, ϵ as the residual error between two iterations i over S array elements, π_S^i, e.g. π_S^1 and π_S^2, and $\epsilon = \pi_S^1 - \pi_S^2$, then $|\epsilon| < e^{-10}$. In layman's terms, this is the way to check how distant two vectors are in absolute terms.

Figure 4 shows a simple example of a CTMC with three states and transition rates. It is used to show the overarching process behind the solution of a CTMC as well as what is the meaning of the results. So, we have modelled an abstraction of a system consisting on three states, then we have observed how it transited among all states and at what rate (e.g. frequency). We then decorated the model with states A, B and C and rates according to the figure.

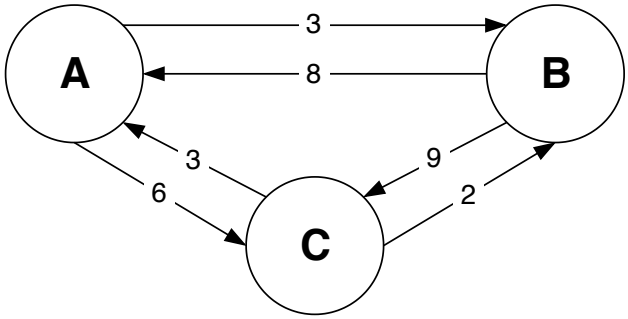

Figure 4: A Markov Chain of three states and its rates.

The objective is to compute the probability of the system being at state A, B or C given those governing rates. The process of solution involves representing this model into a matrix, adjusting it to become an Infinitesimal Generator (in the diagonal), converting it to its Embedded DTMC representation and then using the Power Method (e.g. it consists on multiplying a matrix by itself iteratively) until each line of the matrix is equal to one another (in this case, we have achieved convergence). For this demonstration we will use the MS-Excel spreadsheet, however, any mathematical tool would suffice.

We begin by representing the matrix into R, e.g. the rate matrix (notice that the diagonal contains zeroed values – it does not matter for a CTMC which rate is given to so called self-transitions because they will be *destroyed* in this step).

$$R_{i,j} = \begin{bmatrix} 0 & 3 & 6 \\ 8 & 0 & 9 \\ 3 & 2 & 0 \end{bmatrix}$$

Next, we shall devise the IG named $Q_{i,j}$ for this matrix by summing the rows and assigning the total value (negated) to each corresponding diagonal of $R_{i,j}$ (updated diagonal values are in bold face).

$$Q_{i,j} = \begin{bmatrix} \textbf{-(3+6)} & 3 & 6 \\ 8 & \textbf{-(8+9)} & 9 \\ 3 & 2 & \textbf{-(3+2)} \end{bmatrix} = \begin{bmatrix} \textbf{-9} & 3 & 6 \\ 8 & \textbf{-17} & 9 \\ 3 & 2 & \textbf{-5} \end{bmatrix}$$

For CTMC solution one could work directly on Q then performing $\pi Q = 0$, i.e., translating Q to a set of linear equations and uniformising the results, i.e. summing the final π vector to one ($\sum_{i=0}^{S} \pi_i = 1$).

$$[A, B, C] \times Q = [0, 0, 0] \;\rightarrow\; \begin{cases} -9A + 8B + 3C &= 0 \\ 3A - 17B + 2C &= 0 \\ 6A + 9B - 5C &= 0 \end{cases}$$

The equations can be seen as a balancing system, i.e., an equilibrium based on inward and outward rates among states (rates that are entering must be equal to rates exiting the states).

$$\begin{cases} 9A &= 8B + 3C \\ 17B &= 3A + 2C \\ 5C &= 6A + 9B \end{cases} \begin{array}{l} \texttt{obtained from 6+3 for A (outward rates)} \\ \texttt{i.e. 8+9 out from B} \\ \texttt{3+2 out from C, must be equal to inwards rates from A (6) and B (9)} \end{array}$$

This is a system with three unknown variables. Since we will uniformise the unknown variables in the end of the process, it will not matter to assign a value to one variable, A, B or C. If $C = 1$, then

$$\begin{cases} 9A = 8B + 3 \\ 3A + 2 = 17B \\ 6A + 9B = 5 \end{cases}$$

Solution is done by rearranging unknowns, i.e. $B = \frac{8B+3}{9}$ (first equation), so $3(\frac{8B+3}{9}) + 2 = 17B$ (second equation), and by using this to the next equation, one finds the value of B and then of A, yielding all three unknowns as:

$$\begin{aligned} A &= 0.51937984 \\ B &= 0.20930233 \\ C &= 1 \end{aligned}$$

Where $A + B + C = 1.72868217$. Uniformising those variables (this is step is not necessary if one had added $A + B + C = 1$ in the equations):

$$A = \frac{0.51937984}{1.72868217} = 0.30044843$$

$$B = \frac{0.20930233}{1.72868217} = 0.121076233$$

$$C = \frac{1}{1.72868217} = 0.578475336$$

Which corresponds to the major result of MC, which is the permanence probabilities for every state. For only three unknowns this process is rather easy to be calculated, however if the model has hundreds or thousands (or even millions) of states then more sophisticated tools must be used (e.g. Maple, Mathematica, GNU/Octave).

One could also work with the *Embedded MC* of Q converting it to its DTMC counterpart D to compute the probabilities vector using one of the above solution techniques. Converting this CTMC to its DTMC representation one must use an *Identity Matrix* having order equal to Q (i.e. I_3).

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad I_k = \begin{cases} 1 & \texttt{if } i = j \\ 0 & \texttt{otherwise} \end{cases}$$

For the conversion to succeed, first we must determine the highest value (in absolute terms) called M_{ax} here of the matrix (not surprisingly, this maximum value is always located on the matrix's diagonal). For this example, the element is located at $Q_{2,2} = -17$. The process consists on taking each cell value, discounting the identity and dividing it by the maximum value (which is constant).

$$D_{i,j} = I_k - \frac{Q_{i,j}}{M_{ax}}$$

Meaning that:

$$D = \begin{bmatrix} 1 - \frac{(-9)}{(-17)} & 0 - \frac{3}{(-17)} & 0 - \frac{6}{(-17)} \\ 0 - \frac{8}{(-17)} & 1 - \frac{(-17)}{(-17)} & 0 - \frac{9}{(-17)} \\ 0 - \frac{3}{(-17)} & 0 - \frac{2}{(-17)} & 1 - \frac{(-5)}{(-17)} \end{bmatrix}$$

Which yields the *stochastic matrix* D (not surprisingly, the same used in the DTMC section):

$$D = \begin{bmatrix} 0.470588235 & 0.176470588 & 0.352941176 \\ 0.470588235 & 0.0 & 0.529411765 \\ 0.176470588 & 0.117647059 & 0.705882353 \end{bmatrix}$$

Observe that each line sums to zero and that for the DTMC some states now have self transitions, meaning that a random *surfer* has a probability to remain in that state at every discrete time event.

This concludes solving CTMCs and addressing its output. Note that other techniques not reviewed here could be used such as direct solution methods or similar approaches.

5 Discrete Event Simulation

Simulation aims to *approximate* systems to an executable code. It is used for a plethora of problems ranging from avionics to manufacturing, being successfully employed throughout the years. The idea is to abstract a system intro a process model through a set of simple modelling primitives and devise a codification that is amenable for execution, yielding measures of interest (throughput, response time, hourly resource cost or customised variables or statistics)[10].

Simulation addresses deficiencies present in analytical modelling, specially intricate behaviours, different probability distributions and customised statistics to enhance analysis, to name a few. For instance, in Queueing Networks, one is

[10]For discrete event modelling formalisms such as DEVS (*Discrete Event System Specification*) we direct readers to [30, 31, 32].

restricted to use only the reduced set of available primitives which remarkably limits modelling options. On top of that, one must accept several assumptions that are underline the formalism, for example, in QN there is no possibility to address *jockeying* (i.e. movement between queues), *burst arrivals* (also called *train* arrivals meaning massive simultaneous arrivals in queues), *defecting or balking* (give up staying in the queue) or *renegading* (give up after a while). One is also usually bounded to use exponential or Poisson distribution so the underlying product form equations make sense. Simulation, in this sense, addresses all these aforementioned shortcomings, however, introduces new ones, for example, results are obtained through transient analysis, statistics must be applied afterwards and it could be hard to validate results.

With time, patience and technical ability, one could implement a simulator. In the work "*A History of Discrete Event Simulation Programming Languages*" [33], Richard E. Nance enumerated six fundamental requirements of simulators: pseudo-number generation (enabling to model randomness properties of systems), probability distribution conversions, processing object lists, statistical analysis procedures, reporting and time management mechanisms.

We will address next pseudo-number generation, the overall process of discrete event simulation and tools. The literature on simulation is vast as many tools are available for researchers and practitioners. For example, Arena [34], OMNeT++ [35] and adevs [32] are widely used in many settings yielding actionable results.

We implemented an M/M/1 queue in a software called YACS, its source code is available in Appendix A where it may be redistributed according to GPLv3 licensing conditions. The software uses the principles described next in terms of pseudo-number and randomness as well as the simulation process in general.

5.1 Pseudo-random number generation

The process is called *pseudo-random* because they are not pure random in the sense that arithmetical features were used for their generation. A pure random number is usually nature or physical process based, i.e., lava movement, passing voltage in transistors, and so on. It is hard to predict or compute the next number based on the previous generated one.

Modern pseudo-random generators employ an equation to generate randomness. The algorithm that produces pseudo-random numbers is called *Linear Congruential Generator* (LCG). The generator uses the following recurrence:

$$X_{n+1} = (aX_n + c)\ mod\ M$$

where M is the *modulus* ($m > 0$), a is the *multiplier* ($0 < a < m$), c is the *increment* ($0 <= c < m$) and X_0 is the *seed* (also known as the starting value). Those values are chosen at will, i.e., for `gcc` (GNU C Compiler) they are set as:

$M = 2^{31}$
$a = 1103515245$
$c = 12345$
X_0 is the seed, configured by the user in a `srand` function

Starting at the seed, it generates the next pseudo-numbers (i.e. X_{n+1}) based on X_n. There are two problems with this approach: first, it uses a deterministic method to generate randomness, and second, those constants should be selected to start repeating only after a huge cycle. It is worth mentioning that, knowing all those constants makes any software deterministic, even if calls to random functions (e.g. `rand()`). The implementation of an LCG is available in several languages. For interested readers, the Rosetta Code project offers a variety of implementations[11].

It is interesting to notice that using a constant seed may help debugging simulations as the same sequence of pseudo-random numbers will be generated deterministically.

5.2 Overall simulation process

DES is used to model and analyse systems with state changes in precise moments of simulated time. The simulation process is simple, it starts with the step of zeroing data structures, i.e., counters, global time, event scheduler, output statistics as well as other parameters vital for execution (mainly stop criteria, number of replications, random seed, warm-up definitions and output time scale to name a few).

Another input for any DES is the process model, i.e., the entities inter-arrival time, the tasks (or processes) it must perform, how they are connected altogether, decisions that could take place and involved resources for each task. Also, modellers assign probability distributions for arrivals and task duration as well as routing probabilities for decisions that are embedded in the model through permitted primitives.

[11] Link: `https://rosettacode.org/wiki/Linear_congruential_generator#C`.

Then, it schedules an initial event and enters a loop executing *microcodes* for each event type (e.g. arrivals or departures from queues or other customised behaviour) until stop criteria is met. While it executes, it saves and updates data structures and variables to show to users in the end of the simulation (after all replications). The execution of the microcode of an event could trigger the scheduling of new events, which are inserted in the event list for later processing (ordered by time, so the first one is always selected which speeds up the process). Each replication performs the same set of steps, which are zeroed and stored for every executed one. The final step consists on presenting results to modellers, performance metrics and statistics for the required number of replications.

The simulation kernel of many tools follow more or less the following mechanism shown in Figure 5. The idea is to execute a loop of event microcodes for the replication count (set earlier) until stop criteria according to the event list that must be processed (for a thorough explanation, refer to [36, 37, 38, 39, 27, 40, 41]).

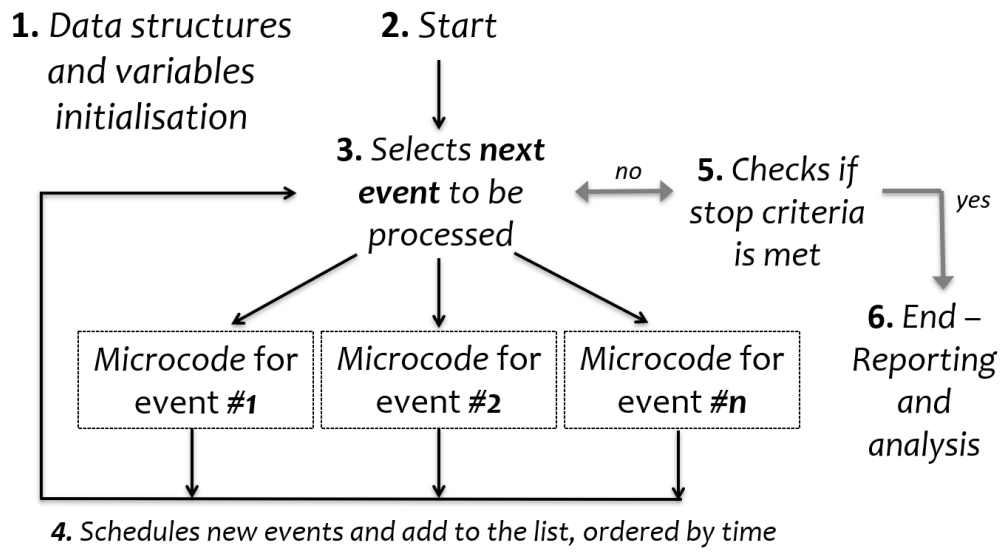

Figure 5: Simulation kernel showing the scheduler and the event microcodes.

Figure 6 shows how the events are executed (E_0 to E_8 in the example). Notice in the figure that time jumps, i.e., it has a discrete behaviour from time to time (t_1 to t_8) where the time offset (i.e. a time interval) is computed allowing the simulator to store how long it took for processing each event and which resources were used.

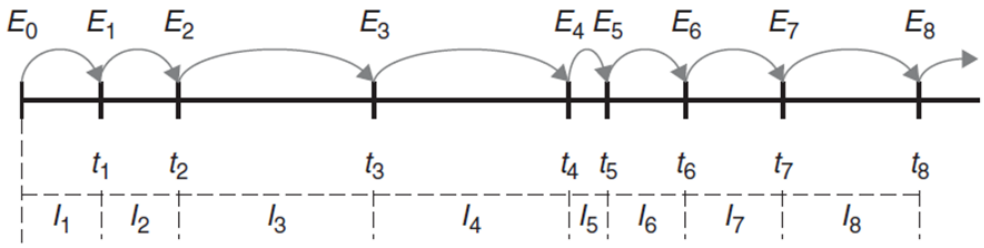

Figure 6: Discrete events and relation to time jumps throughout simulation process.

This ends the process of executing a discrete event simulator. It is noticeable the easiness of the process as a whole where it is quite effortlessly to implement a customised simulator targeted for specific customisation.

6 Concluding remarks

Systems-of-systems, either *Information Technology* servers, Internet infrastructure, sensors, IoT or other Cyber-Physical Systems permeate many societal levels. They employ different technologies to provide services for users. The point of this work is focus on the *quality* aspect of the service they deliver. It is now more crucial than ever to address quantitative properties of these system-of-systems, as well as to quantify the influence of their interaction. However,

before considering Performance Evaluation, one must familiarise with concepts from this domain to extract most value when addressing analysis.

The mathematical background required in Performance Evaluation or Markov Chains to that effect usually hinders adoption or simple understanding of key principles that could be abstracted. This work aimed to present an overview of those techniques and present reader with a reasonable framework to enlarge the understanding of its uses in system evaluation.

Important modelling approaches and solution techniques were presented, where modellers could choose the one best suited for the problem at hand. The idea is to use the framework set here and allow modellers to define their own parameters, states, transitions, times and rates to devise a performance model suited for posterior analysis. One could use Markov Chains (discrete or continuous) or discrete event simulation in a seamless fashion, maybe using the solution offered here (e.g. the code for MCs or for the simulation). Perhaps the most significant application of Markov Chains runs inside the core of Google's search engine for millions of daily searches. This has been the subject of a considerable body of work in the field of Information Retrieval using MC and (fast) transient analysis [42].

Finally, there are other methods and techniques that were not addressed here such as *phase type* models (where different probabilities distributions could be considered), *Agent Based Simulation* or *Parallel DES* implementations, for instance, among many other. The reason is that this work aimed as a *brief* introduction to the broad area of performance evaluation.

References

[1] IEEE 1012-2016. IEEE Standard for System, Software, and Hardware Verification and Validation. pages 1–260, 2012.

[2] ISO/IEC/IEEE 24765:2010. Systems and software engineering – Vocabulary. pages 1–418, 2010.

[3] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[4] Daniel A. Menasce, Virgilio Almeida, and Lawrence W. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[5] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall PTR, 2001.

[6] ISO 25065:2019. Systems and software engineering – Software product Quality Requirements and Evaluation (SQuaRE) — Common Industry Format (CIF) for Usability: User requirements specification. pages 1–20, 2019.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[8] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[9] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[10] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[11] Arnd Hartmanns and Holger Hermanns. In the quantitative automata zoo. *Science of Computer Programming*, 112:3–23, 2015.

[12] David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005.

[13] Gianfranco Balbo. Introduction to stochastic petri nets. In *School organized by the European Educational Forum*, pages 84–155. Springer, Berlin, Heidelberg, 2000.

[14] Brigitte Plateau and Karim Atif. Stochastic automata network of modeling parallel systems. *IEEE Transactions on Software Engineering*, (10):1093–1108, 1991.

[15] Amy N. Langville and William J. Stewart. The Kronecker product and Stochastic Automata Networks. *Journal of Computational and Applied Mathematics*, 167(2):429–447, 2004.

[16] Jane Hillston. *A compositional approach to performance modelling*, volume 12. Cambridge University Press, 2005.

[17] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal methods in system design*, 15(1):7–48, 1999.

[18] Benjamin Camus, Thomas Paris, Julien Vaubourg, Yannick Presse, Christine Bourjot, Laurent Ciarletta, and Vincent Chevrier. Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware. *Simulation*, 94(12):1099–1127, 2018.

[19] Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, Peter Fritzson, Jörg Brauer, Christian Kleijn, Thierry Lecomte, Markus Pfeil, Ole Green, Stylianos Basagiannis, and Andrey Sadovykh. Integrated tool chain for model-based design of cyber-physical systems: The INTO-CPS project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6. IEEE, 2016.

[20] James Nutaro, Phani Teja Kuruganti, Laurie Miller, Sara Mullen, and Mallikarjun Shankar. Integrated hybrid-simulation of electric power and communications systems. In *2007 IEEE Power Engineering Society General Meeting*, pages 1–8. IEEE, 2007.

[21] William J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[22] James Robert Norris. *Markov chains*. Number 2. Cambridge University Press, 1998.

[23] Olle Häggström. *Finite Markov chains and algorithmic applications*, volume 52. Cambridge University Press, 2002.

[24] Gunter Bolch, Stefan Greiner, Hermann De Meer, and Kishor S. Trivedi. Queueing networks and markov chains: modeling and performance evaluation with computer science applications, 2006.

[25] Allan Lee Scherr. *An analysis of time-shared computer systems*, volume 71. Mit Press Cambridge, Mass., 1967.

[26] William J. Stewart. Performance modelling and markov chains. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 1–33. Springer, 2007.

[27] William J. Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton university press, 2009.

[28] Kishor S Trivedi and Andrea Bobbio. *Reliability and availability engineering: modeling, analysis, and applications*. Cambridge University Press, 2017.

[29] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[30] Arturo I. Concepcion and Bernard P. Zeigler. DEVS formalism: A framework for hierarchical model development. *IEEE Transactions on Software Engineering*, 14(2):228–241, 1988.

[31] Bernard P. Zeigler. Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation*, 49(5):219–230, 1987.

[32] Alexander Muzy and James J. Nutaro. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling & Simulation (OICMS)*, pages 273–279, 2005.

[33] Richard E. Nance. A history of discrete event simulation programming languages. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 149–175, New York, NY, USA, 1993. ACM.

[34] Manuel D. Rossetti. *Simulation modeling and Arena*. John Wiley & Sons, 2015.

[35] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and . . . , 2008.

[36] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-event system simulation*, volume 3. Prentice hall Upper Saddle River, NJ, 1996.

[37] Michael Pidd. *Computer simulation in management science*, volume 4. Wiley Chichester, 1998.

[38] Averill M Law and W David Kelton. *Simulation modeling and analysis*, volume 3. McGraw-Hill New York, 2000.

[39] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of modeling and simulation*. Academic press, 2000.

[40] John A. Sokolowski and Catherine M. Banks. *Modeling and simulation fundamentals: theoretical underpinnings and practical domains*. John Wiley & Sons, 2010.

[41] Bernard P. Zeigler, Hessam S. Sarjoughian, Raphaël Duboz, and Jean-Christophe Soulié. *Guide to modeling and simulation of systems of systems*, volume 516. Springer, 2013.

[42] Amy N Langville and Carl D Meyer. *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2011.

A YACS – *Yet Another Computer Simulator* – source code

YACS implements a *Discrete Event Simulation* of an M/M/1 queue. One defines the arrival rate of entities and its service rate (more information on file `yacs.c`). The stop criteria is set to stop after 10.000 minutes of operation, however the user could select a number of events to process before stopping (those options are commented on the source code). **It is worth mentioning that this software has not been validated.**

A.1 Licence – GPLv3 header

The software is open-source under GPLv3 license. Each file has the following header:

```
/*
This is YACS, Yet Another Computer Simulation software, using DES to simulate an M/M/1 queue.
Copyright (C) 2019  Ricardo M. Czekster
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software Foundation,
Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301  USA
*/
```

A.2 Source code

YACS was implemented in *The C Programming Language*, for efficiency. The following source code listing shows the implementation of the solution with a standard header file (`defs.h`), the main file (`main.c`) that calls the simulator, the simulator details (`yacs.c`), an auxiliary source code that implements a linked list (for ordered insertion in the scheduling list, called `linkedlist.c`) and a file with random number generators (mostly exponentially and uniformly distributed variables in `randomgenerators.c`).

---

defs.h

```
#ifndef __DEFS__
#define __DEFS__

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/// List of events
#define ARR 0  // arrival event
#define DEP 1  // departure event

/// Properties
// stop criteria
#define SC_DURATION 0
#define SC_EVENTS   1

static int cid = 0;

/**
 * A node for the double linked list (of events).
```

```
 */
typedef struct {
   int id;
   double duration;
   double beginqueue;
   int type; // type of the event: arrival or a departure
   struct node* next;
   struct node* prev;
} node;

/**
 * Simulation properties.
 */
typedef struct {
   int stopcriteria; // look constants SC_DURATION and SC_EVENTS (above)
   double maxtime;   // if stopcriteria == SC_DURATION, then maxtime should be inspected
   int maxevents;    // if stopcriteria == SC_EVENTS, then maxevents should be inspected
} properties;

/**
 * Simulation main structure.
 */
typedef struct {
   double CLOCK;      // global time
   node* eventlist;
   properties* props; // simulation properties (see above)
} simulation;

/**
 * Function prototypes for the entire project.
 */

/////// linkedlist.c
extern int listsize(node** head);
extern void position_insert(node** head, int pos, int value);
extern void ordered_insert(int t, double clock, double value, node** head);
extern node* dequeue(node** head);
extern void dealloc(node** head);
extern void show(node** head);

/////// randomgenerators.c
extern double nextexponential(double rate);
extern double nextuniform01();
extern double nextuniform(double a, double b);

/////// yacs.c
extern void simulate();
extern void init(simulation* s);
extern void dealloc_simulation(simulation* s);

/**/

#endif // __DEFS__
```

---

randomgenerators.c

```
#include "defs.h"
```

```
/**
 * Returns the next exponential random variable according to a given rate (as parameter).
 * It converts a uniformly random variable to an exponential random variable.
 */
double nextexponential(double rate) {
   double unif;
   do {
      unif = rand() / (double) RAND_MAX; // a uniformly distributed variable
   } while (unif == 0.0 || unif == 1.0);
   //return (-1 * log(1 - unif) * (1.0 / rate));
   return (-log(unif) / rate); // from William Stewart's book (2009) 'Probability, Markov Chains,
                               //   Queues, and Simulation: The Mathematical Basis of
                               //   Performance Modeling'
}

/**
 * Returns the next uniform between 0 and 1.
 *
 */
double nextuniform01() {
   return (nextuniform(0, 1));
}

/**
 * Returns the next uniform between a and b.
 * If b >= a, then returns a number between 0 and 1.
 */
double nextuniform(double a, double b) {
   double unif = rand() / (double) RAND_MAX;
   if (a < b)
      unif = unif * (b - a) + a;
   return (unif);
}
```

---

main.c

```
#include "defs.h"

int main() {

   simulation *sim = (simulation*) malloc(sizeof(simulation));
   init(sim);
   simulate(sim);

   dealloc_simulation(sim);
   free(sim);

   return EXIT_SUCCESS;
}
```

---

yacs.c

```
#include "defs.h"

void init(simulation* s) {
   srand(time(NULL));
   s->CLOCK = 0.0;
```

```
    s->eventlist = NULL;
    s->props = (properties*) malloc(sizeof(properties));
    s->props->stopcriteria = SC_DURATION;
    s->props->maxtime = 100000.0;  // simulation target time (10.000 min)
    // if you would want to use number of events as stop criteria, uncomment the following lines
    //s->props->stopcriteria = SC_EVENTS;
    //s->props->maxevents = 100;           // e.g. 100 events
}

void simulate(simulation* s) {
    int numevents = 0;       // number of events
    double duration;
    double utilization;      // perf metrics
    double population;       // ""
    int entities = 0;
    int numarrivals = 0;     // counter for arrivals in the system
    int numdepartures = 0;   // counter for departures in the system
    int processed = 0;
    int departed = 0;
    int servers = 1;         // ---> change the number of servers
    node* nextevent;
    double elapsedtime;
    double busytime = 0.0;
    double queuelength = 0.0; // counter for queuelength (number waiting)
    double queueing = 0.0;
    double totaltime = 0.0;
    double servicetime = 0.0;
    double saveclock = 0.0;
    double mu = 0.0;
    FILE *fp;                 // file for logging
    // rates
    double arr_rate = 60.0/11.0; // rates over one hour
    double dep_rate = 60.0/(8.0/(double)servers); // rates over one hour - increases velocity for more se

    // start the scheduler with an arrival schedule to happen according to the arrival rate
    duration = nextexponential(arr_rate);  // according to the arrival rate
    ordered_insert(ARR, s->CLOCK, duration, &(s->eventlist)); // schedule an arrival

    // creates a log file
    fp = fopen("output.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        exit(EXIT_FAILURE);
    }

    // start consuming events from the event list (this is the DES core implementation)
    do {
        nextevent = dequeue(&(s->eventlist)); // dequeue next node from the event list
        elapsedtime = nextevent->duration - s->CLOCK;
        if (numevents % 1000 == 0)
            printf("simulation clock: %f events: %d\r", s->CLOCK, numevents);

        if (entities != 0)
            busytime = busytime + elapsedtime;
        saveclock = s->CLOCK;
        s->CLOCK = nextevent->duration; // updates the simulation clock (discrete time)
        // microcode for each possible event within the DES
        switch (nextevent->type) {
            case ARR: // arrivals
```

```
            queuelength = queuelength + entities;
            entities = entities + 1;
            if (entities <= 1) {
               duration = nextexponential(dep_rate);  // according to the departure rate
               ordered_insert(DEP, s->CLOCK, duration, &(s->eventlist)); // schedule a departure
               mu = mu + duration;
               processed = processed + 1;
               //queueing = queueing + (s->CLOCK - nextevent->beginqueue);
            }
            // schedule the next arrival
            duration = nextexponential(arr_rate);
            ordered_insert(ARR, s->CLOCK, duration, &(s->eventlist));
            numarrivals = numarrivals + 1; // count only arrivals
            break;
         case DEP: // departures
            entities = entities - 1;
            if (entities >= 1) { // schedule the departure for the next entity (starts serving the next)
               duration = nextexponential(dep_rate);  // according to the departure rate
               ordered_insert(DEP, s->CLOCK, duration, &(s->eventlist)); // schedule a departure
               departed = departed + 1;
               queueing = queueing + (saveclock - nextevent->beginqueue);
            }
            numdepartures = numdepartures + 1; // count only departures
            break;
      }
      numevents = numevents + 1;
      // free the node
      free(nextevent);
   } while ((s->props->stopcriteria == SC_DURATION && s->props->maxtime >= s->CLOCK) ||
            (s->props->stopcriteria == SC_EVENTS && s->props->maxevents >= numevents));

   // reporting
   printf("simulation clock: %f events: %d\r", s->CLOCK, numevents); // flushes previous \r
   printf("\nEvents: %d\n", numevents);
   printf("Busy Time: %f\n\n", busytime);
   printf("Arrivals: %d\n", numarrivals);
   printf("Departures: %d\n", numdepartures);
   printf("Processed departures: %d\n", processed);
   printf("Servers: %d\n", servers);
   printf("Arrival   [time=%f -> rate=%f]\n", 60.0/arr_rate, arr_rate);
   printf("Departure [time=%f -> rate=%f]\n\n", 60.0/dep_rate, dep_rate);
   // performance metrics
   printf("\nPerformance metrics\n");
   utilization = busytime / s->CLOCK;
   printf("\tUtilization: %f\n", utilization);
   population = queuelength / (double)numarrivals;
   printf("\tQueue length: %f\n", population);
   totaltime = (queueing + mu) / (double)processed;
   queueing = queueing / (double)processed;
   servicetime = mu / (double)processed;
   printf("\tavg total time: %f\n", totaltime*60);
   printf("\tavg queueing time: %f\n", queueing*60);
   printf("\tavg service time: %f\n", servicetime*60);
   fclose(fp);
}

/**
 * Dealloc the entire simulation and its structures
 */
```

```
void dealloc_simulation(simulation* s) {
   dealloc(&(s->eventlist)); // frees the remaining nodes from the dynamic list
   free(s->props);
}
```

---

linkedlist.c

```
#include "defs.h"

int listsize(node** head) {
   int c = 0;
   node* aux = *head;
   while (aux!=NULL) {
      c = c + 1;
      aux = aux->next;
   }
   return c;
}

/**
 * Insert element on position.
 */
void position_insert(node** head, int pos, int value) {
    int i, tam;
    tam = listsize(*head);
    node* aux = *head;
    node* target;
    if (pos <= tam && pos >= 0){
        target = (node*) malloc(sizeof(node));
        target->duration = value;
        if (pos == 0){
            target->next = *head;
            *head = target;
        } else {
            for (i=0; i < pos - 1; i++) {
                aux = aux->next;
            }
            target->next = aux->next;
            aux->next = target;
        }
    }
}

/**
 * Insert element by ascending order considering 'value' parameter.
 */
void ordered_insert(int t, double clock, double value, node** head) {
   node* aux = *head;
   node* elem;
   node* next;
   elem = (node*) malloc(sizeof(node));
   elem->duration = clock + value; // T + value
   elem->beginqueue = clock;
   elem->type = t;
   cid = cid + 1;
   elem->id = cid;
   if (*head == NULL) {
      elem->next = NULL;
```

```
      elem->prev = NULL;
      *head = elem;
   } else {
      if (aux->duration >= elem->duration) {
         elem->next = *head;
         elem->prev = NULL;
         (*head)->prev = elem;
         *head = elem;
      } else {
         while (elem->duration > aux->duration && aux->next != NULL) {
            next = aux->next;
            if (elem->duration < next->duration)
               break;
            aux = aux->next;
         }
         elem->next = aux->next;
         elem->prev = aux;
         aux->next = elem;
         next = elem->next;
         if (elem->next != NULL)
            next->prev = elem;
      }
   }
}

/**
 * Consume the first element of the list.
 */
node* dequeue(node** head) {
   node* aux = *head;
   if (*head != NULL)
      *head = (*head)->next;
   return aux;
}

void dealloc(node** head) {
   node* aux = *head;
   node* target = aux;
   while (target != NULL) {
      aux = aux->next;
      free(target);
      target = aux;
   }
   if (*head != NULL)
      free(*head);
}

void show(node** head) {
   node* aux = *head;
   printf("List contents:\n");
   while(aux != NULL) {
      printf("%d -> %2.6f\n", aux->type, aux->duration);
      aux = aux->next;
   }
   printf("\n");
}
```