The Journal of Systems & Software 191 (2022) 111385

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Mutation-based analysis of queueing network performance models ☆

Thomas Laurent [a,*], Paolo Arcaini [b], Catia Trubiani [c], Anthony Ventresque [a]

[a] Lero & University College Dublin, Dublin, Ireland
[b] National Institute of Informatics, Tokyo, Japan
[c] Gran Sasso Science Institute, L'Aquila, Italy

ARTICLE INFO

Article history:
Received 22 November 2021
Received in revised form 13 May 2022
Accepted 27 May 2022
Available online 5 June 2022

Keywords:
Queueing Networks
Mutation
Model-based performance analysis

ABSTRACT

Performance models have been used in the past to understand the performance characteristics of software systems. However, the identification of performance criticalities is still an open challenge, since there might be several system components contributing to the overall system performance. This work combines two different areas of research to improve the process of interpreting model-based performance analysis results: (i) software performance engineering that provides the ground for the evaluation of the system's performance; (ii) mutation-based techniques that nicely supports the experimentation of changes in performance models and contribute to a more systematic assessment of performance indices. We propose mutation operators for specific performance models, i.e., queueing networks, that resemble changes commonly made by designers when exploring the properties of a system's performance. Our approach consists in introducing a mutation-based approach that generates a set of mutated queueing network models. The performance of these mutated networks is compared to that of the original network to better understand the effect of variations in the different components of the system. A set of benchmarks is adopted to show how the technique can be used to get a deeper understanding of the performance characteristics of software systems.

1. Introduction

Evaluating the performance characteristics of software systems is a problem recognised by both researchers and practitioners as of high relevance, to the extent that performance has emerged, among other extra-functional properties, as contributing to the *new correctness* (Harman and O'Hearn, 2018). Indeed, if performance requirements are not met, negative consequences may arise, e.g., damaged customer relations resulting in economic loss. Concrete examples of performance requirements are: a system response time not larger than 10 s, a service throughput not lower than 5 requests/second, and a utilisation of hardware devices not higher than 90%. To verify the fulfilment of these requirements, many techniques have been developed in the software performance engineering community (Smith and Williams, 2002; Petriu et al., 2012; Cortellessa et al., 2011; Bondi, 2014), and there is a growing interest in the early validation of requirements to prevent design errors from propagating to the end system (Haskins et al., 2004).

The complexity of the problem is exacerbated when considering that the openness, heterogeneity, and versatility of modern and software-intensive systems entail the specification of different types of uncertainties, all having an impact on the performance behaviour of the system. Uncertainties may span several dimensions; for instance, it is very unlikely that the workload is uniquely determined, and several fluctuations may be observed when the system is in operation. Other types of uncertainties may manifest as: (i) operational profile, i.e., users requiring different services with uncontrolled temporal ordering; (ii) service demands, in case software services require more/less resources depending on load conditions; (iii) hardware settings, in case some failures happen, and software needs redeployment on different platforms.

To deal with different types of system uncertainties, performance models are of key relevance, since they allow to reflect uncertainties in the specification of their input parameters (Aleti et al., 2018; Pinciroli et al., 2017; Mishra and Trivedi, 2011). This work focuses on *Queueing Networks* (*QNs*) (Lazowska et al., 1984; Kleinrock, 1975) as target performance model, one of many such modelling techniques (Koziolek, 2010). QNs are a well-assessed formalism in the software performance engineering community; moreover, they are recognised as especially suited to resource-sharing contexts (Balsamo and Marzolla, 2005; Smith et al., 2010) and as having good prediction accuracy when analysing real

☆ Editor: Burak Turhan.

* Corresponding author.
E-mail addresses: thomas.laurent@ucd.ie (T. Laurent), arcaini@nii.ac.jp (P. Arcaini), catia.trubiani@gssi.it (C. Trubiani), anthony.ventresque@ucd.ie (A. Ventresque).

https://doi.org/10.1016/j.jss.2022.111385

systems (Urgaonkar et al., 2005; Casale et al., 2016; Dipietro et al., 2016; Incerto et al., 2021). Furthermore, they have been shown to properly model the uncertainties inherent to system design (Antonelli et al., 2020).

Interpreting performance analysis results is critical, since simulation results only give a snapshot of the system's performance but do not provide any hint of *why* these results are obtained, and their root causes. System designers need to know which parts of the system contribute to negative performance results; namely, they are interested in identifying which system elements lead to bad system performance, or are unnecessarily set to expensive software/hardware settings, increasing the overall system cost. To understand root causes of performance issues, system designers usually need to manually modify the network (e.g., by increasing or decreasing the number of servers in a node) to check whether these changes affect the system performance. Such an approach has several problems: it requires performance-based expertise, it is tedious, and it may also be error-prone. Thus, this process should be automated.

The goal of this paper is to provide an automated framework that can assist designers in assessing the performance of a system and sizing its resources, by identifying performance bottlenecks, challenging workloads, and possibilities for refactoring/design changes. We build such a framework by joining the forces of two different areas of research. First, *software performance engineering* techniques are adopted as the ground for the performance evaluation of software systems. Second, we take inspiration from *mutation analysis* (Papadakis et al., 2019), and apply changes to performance models to automatically consider alternative designs in terms of performance and cost trade-off analysis.

This work introduces a novel *mutation-based approach* that generates a set of mutated queueing network models, and uses them to better understand the performance of the original system and of alternative designs. To this end, we define *mutation operators* resembling the network modifications that are usually performed by designers when tuning the performance of the system and assessing cost. Moreover, we define a way to compare the performance results produced by these mutants with those of the original model under the same workload.

The framework helps get a better understanding of the system's performance and of the effect of the different design elements. For example, it can highlight elements central to the overall performance of the system, or that the system is over-equipped.

The approach was applied to a benchmark set of five QNs developed by practitioners or proposed in the literature. Experimental results confirm the relevance of the proposed framework in understanding the involvement of each of the system's components, and their effect on the different aspects of the performance.

To summarise, the main contributions of this work are:

- a mutation-based approach for analysing the performance results from queueing network models, and the associated system costs required to meet them — see Section 3.2;
- the specification of *mutation operators* that resemble the typical design changes that can be performed to run a trade-off analysis between cost and performance — see Section 3.3;
- a framework implementing and automating the application of the approach – see our publicly available replication package (Laurent et al., 2022);
- a set of experiments illustrating the benefits of applying the approach to support designers in the understanding of performance and cost evaluation vs model components — see Section 4.

The paper is organised as follows. Section 2 presents background information on queueing networks and describes an illustrative example QN model. Section 3 explains the proposed approach and provides details on the mutation operators. Section 4 presents the experiments we conducted to evaluate the proposed approach, and Section 5 provides a critical discussion of the work. Section 6 discusses some threats that may affect the validity of the approach and steps taken to mitigate them. Finally, Section 7 reviews related work, and Section 8 reports concluding remarks and possible research directions for future work.

2. Background and illustrative example

We here provide background concepts on which this work is built. Section 2.1 introduces the minimal necessary definitions expressing Queueing Network models and background on mutation analysis. Section 2.2 illustrates an example of QN model.

2.1. Background

Queueing Networks (QNs) have been widely applied to represent and analyse resource sharing systems (Kleinrock, 1975). A QN model can be defined as a collection of interacting *processing nodes* representing system resources, a set of *customer classes* that refer to the different system requests, and *structural nodes* representing the flow of requests.

Processing nodes are of two types. The first type are *service centres*, which are composed of a *Server* and a *Queue*. Queues can be characterised by a finite or an infinite size. Each server, contained in a service centre, picks the next job from its queue (if not empty), processes it, and routes the processed request to another node. The second type of processing nodes are *delay centres*, that differentiate from the service centres in not having an associated queue.

Service and delay centres are connected through *links* that form the network topology. A *routing node* can be introduced to route customer classes to different branches $(b_1, \ldots, b_n)$ according to some given probabilities $(p_1, \ldots, p_n)$, assuming that $p_i \in [0, 1]$ and $\sum_{i=1}^{n} p_i = 1$.

In other words, the QN representation is a direct graph whose nodes are service/delay centres and their connections are represented by the graph edges. *Jobs* go through the graph's edge set on the basis of the behaviour of customers' service requests. Moreover, there might be *fork nodes* that are used to express parallelism, i.e., one task is split into multiple activities executed in parallel and synchronised through the *join node*. To regulate the parallelism, there exist *finite capacity regions* that explicitly express the maximum number of requests that can run in parallel.

The time spent in every processing node by each request is modelled by probability distributions, e.g., exponential or deterministic distributions. Delay centres are described only by a service time that denotes how long jobs are delayed before proceeding in further delay or queueing centres. Service centres also include the specification of service times needed to process the different types of requests, along with the policy to manage the requests waiting in the queue, e.g., first-come-first-served (FCFS). The incoming workload can be modelled as *open* (i.e., specified by an arrival rate λ) or *closed* (i.e., a constant number N of requests specified as the population size). In case of open workload, requests are generated by *source nodes* connected with links to the rest of the QN, and terminate in *sink nodes* when all tasks have been performed.

Due to system uncertainties (such as fluctuating workload, operational profile, resource demand of services, service time of hardware devices), setting the input parameters of QN models has been recognised as particularly critical (Trubiani et al., 2013),

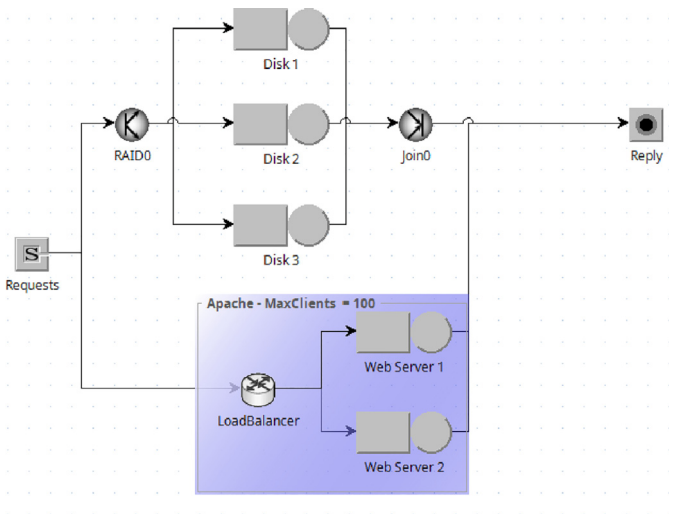

Fig. 1. QN model example (from ShicaoUW (2017)).

and recently some approaches have been defined that foster sensitivity analysis (e.g., multiple load tests) as fundamental in the process of software performance engineering (Bondi, 2014). The approach described in this work takes inspiration from sensitivity analysis, in that it uses performance results in order to better understand how the system reacts to different changes.

Mutation analysis is a test criterion that introduces some changes mimicking faults in a system under test, i.e., it mutates the systems under test, and measures the capability of the tests to detect these changes, or *mutants* (DeMillo et al., 1978; Papadakis et al., 2019). If the tests do not detect the mutants then a gap in the test suite has been highlighted. Mutants normally aim at changing the semantics of the system under test by modifying it, by syntactically altering the source code of the application, or the model that is being used for testing. However, some mutants – called *equivalent mutants* – remain semantically equivalent to the original system, no test could ever detect them. Equivalent mutants are usually seen as a hindrance in mutation analysis. It seems like there is a gap in the test suite, even though there is none. However, this work relies on equivalent mutants, as we want to compare the performance of design alternatives of a system that achieve the same functionality, in order to better understand the effect of each design element on the performance results. This means that we are interested in changes to the queueing network performance model that do not alter what the modelled system does (in terms of processing operational steps), but may affect the overall system performance. For instance, the addition of a processing node can indicate a further service to be managed by the software system, and the resulting QN model is not functionally equivalent. As opposite, changing the policy by which queued requests are served can speedup or delay the processing, but the system functionality is not affected.

2.2. Illustrative example

Fig. 1 provides, as an illustrative example, a publicly available QN model (ShicaoUW, 2017). There is a source node, namely *Requests*, that represents an open workload of jobs coming into the system. Two different types of requests have been defined: (i) *Class0* follows an hyper exponential distribution, and (ii) *Class1* presents an exponential distribution. Both classes are routed to the *RAID0* fork node and the finite capacity region called *Apache — MaxClients = 100*. Each request routed to the *RAID0* fork node is transformed to three tasks that are forwarded to the next queueing network centres, i.e., *Disk 1*, *Disk 2*, and *Disk 3*, each showing a certain service time distribution. The completion of forked tasks is regulated by the join node *Join0*. This latter node is connected to the *Reply* sink node that determines the completion of such requests.

Each request routed to the finite capacity region follows its rules; specifically, the region capacity is set to 100, that means no more that one hundred requests are allowed to reach the *LoadBalancer* routing station at a time. A strategy of routing is set for each job class; for instance, *Class0* joins the shortest queue, and *Class1* follows the shortest response time criterion. Two queueing centres (i.e., *Web Server 1* and *Web Server 2*) are connected to the routing station in order to balance the load that terminates in the *Reply* sink node.

3. Mutation-based analysis of queueing networks

This section describes our main contribution, i.e., a *mutation-based approach to performance analysis* by means of Queueing Networks. First, Section 3.1 introduces performance analysis and the typical use case in which it is applied. Then, Section 3.2 provides an overview of the proposed mutation-based analysis approach, and the following sections detail the mutation operators (Section 3.3), the performance metrics (Section 3.4), the simulation process (Section 3.5), and the methods for comparing the original and mutated model's performance (Section 3.6).

3.1. Current practice in model-based performance analysis

Queueing Networks (QNs) are used to analyse the system performance, often through simulation (Lazowska et al., 1984). A system can be modelled with a QN and then different workloads can be simulated for such a system. Several *performance metrics* can be collected during the simulation to represent the performance of the system. Performance analysis can fulfil multiple goals. A typical use case is *Operational Profile Assessment* (OPA): knowing the workloads the system will encounter, and the performance requirements that it needs to meet, OPA checks whether the system meets the requirements, and can help find possible modifications of the system, either to meet violated requirements, by adding additional resources, or to minimise costs of the system, by removing unnecessary resources. This work focuses on OPA as a use case of Queueing Network, and thus assumes the system's typical workloads are known.

The current practice involves substantial manual (subject to be error-prone) analysis by performance experts in order to answer questions like "which elements in the system are responsible for performance issues?", "which elements are under-utilised (i.e., they are oversized)?", or "how do different workloads affect performance results?". The next section describes a mutation-based approach to analyse these results in a more systematic way, in order to more easily gain insights on the performance and the cost of the system.

3.2. Overview of the approach

This work tackles the problems related to model-based performance analysis stated above. The proposed approach is inspired by mutation analysis (Papadakis et al., 2019), where modifications are seeded into a system to better understand its code, and to find tests that capture these modifications. Although mutation analysis is usually applied to seed faults in the system to check whether the tests are strong enough to capture the faults, a different line of work also uses mutation to *repair* the system (Gazzola et al., 2019), i.e., seeding modifications that could remove some faults

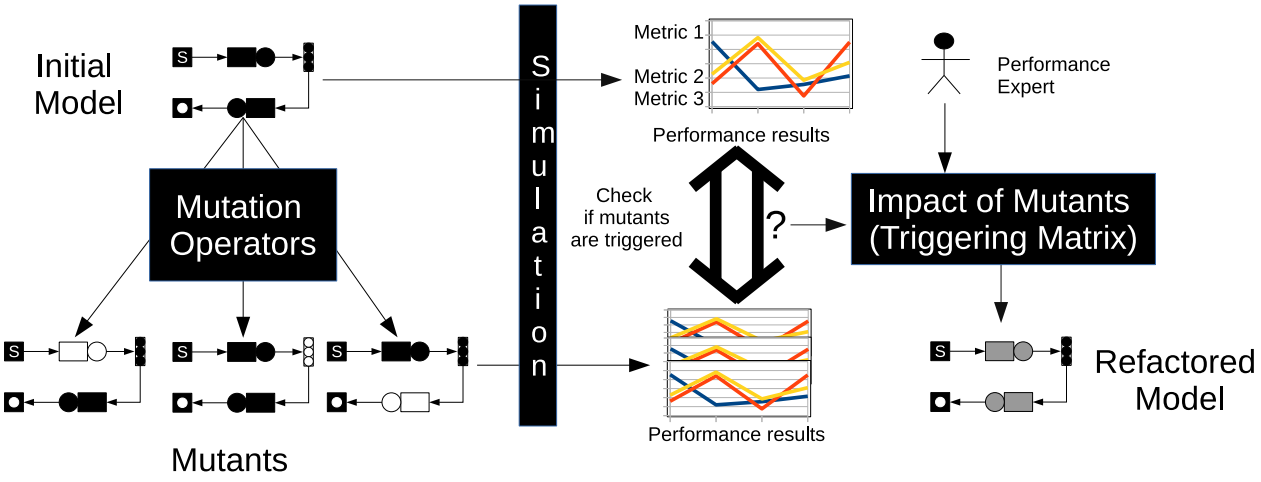

Fig. 2. Mutation-based analysis of queueing networks.

or improve the system. In our approach, we embrace both usages of mutation analysis, and we use them together.

Namely, we propose a mutation-based approach for performance analysis that introduces modifications (e.g., larger or smaller queue size, different queueing strategy, etc.) in a Queueing Network model to better understand the system's performance. Introducing these variations in performance models and studying their effect on the system's performance helps explain the contribution of the system's different elements to the overall performance, and highlights opportunities for refactoring the system, either by increasing or decreasing its used resources. To the best of our knowledge, this work is the first application of mutation-based concepts to better understand performance models and analysis results.

Fig. 2 gives an overview of the mutation-based approach to performance analysis. As described above, a QN model of the system under analysis and a set of workloads are traditionally used for performance analysis. These elements are also present in the proposed mutation-based approach, where workloads are simulated (as described in Section 3.5) on the model to produce performance results over the different metrics described in Section 3.4. The mutation-based approach differs from the traditional approach in that not only the original system's model is used. *Mutant models* are generated from the system's model by applying the mutation operators described in Section 3.3, that introduce modifications in the model. The workloads are simulated on these mutant models as well and produce other performance results.

The performance of the original model and of the mutant models are then compared using the method described in Section 3.6. If the performance results of the original model and of a mutated model are different, then the workloads show the difference in the mutant, and we say the mutant was *triggered*. Instead, if the results are similar, it means that the modification is irrelevant for the workloads. At this stage, a *triggering matrix* can be produced, showing which workloads trigger which mutants (and with regards to which metric).

This matrix is then analysed for insights on the system's performance and the workloads used. The output of the mutation process brings valuable insights, no matter the effect of the mutants on the performance results. Depending on the effect observed, the insights can be of a different nature.

On the one hand, if a mutant is not triggered, then it shows it does not impact the different performance metrics for any of the workloads considered. If the system designer wants to modify (usually improve) the system's performance, then they know that the type of modification introduced by the mutant will have no effect, and they can focus their redesigning efforts on other parts of the system. However, if the goal of the designer is to save costs, and if the mutant represents a cost-saving modification (e.g., a smaller queue), then they know that the type of change represented by the mutant does not impact the performance, and is a viable cost-saving modification.

On the other hand, if a mutant is triggered by a workload, then it shows it had an effect on the system's performance under these circumstances. If the system designer wants to modify the system's performance, then this result highlights a component of the system they can focus on to achieve this. They can then examine the details of the performance results achieved by the mutant model to determine if they want to apply the type of change represented by the mutant. For example, the designer can check whether the mutant improved or degraded performance, and w.r.t. which metrics (one metric could be improved while another is degraded), and if the changes to the system's performance can justify the change in the cost of the system.

Although some cases, such as a mutant implying a reduction in the cost of the system and having a positive effect on its performance, would be very easy to analyse, some degree of expertise is still required to balance the overall trade-offs involved in designing systems. However, the mutation-based approach systematically and automatically explores possible changes to the system, a task that would usually be done manually, only relying on "rules of thumb" and the designer's domain knowledge. Furthermore, the approach provides a condensed view of the effect of all changes (the triggering matrix), that highlights particular changes the designer can focus on to achieve their goals, preventing waste of the expert's time on dead ends.

3.3. Mutation operators

This section describes possible mutation operators for QNs. We focus on common-practice design changes that have been applied to performance models (Petriu, 2021; Xu, 2012) and give insights into the causes for poor performance and how to fix them (e.g., by removing bottlenecks). The goal of mutation operators is to cover well-assessed system changes; these changes are motivated by users that can benefit from the understanding of the system design and its correlated performance fluctuations. Hence, this work investigates three design elements whose configuration can have an impact on different performance results, and also require diverse costs when implementing the system. Namely, these are:

Queue size. It relates to the capacity of queueing stations to host a number k of requests waiting to be served. The mutation operators focus on creating finite queues, in order to use the request drop rate as an indicator of busy times for resources.

Queueing strategy. It refers to the policy of queueing stations to handle requests waiting to be served. There are multiple options that can be selected from, e.g., (i) (ii) First Come First Served (FCFS), i.e., customers are served in the same order in which they arrive at the station; (iii) Last Come First Served (LCFS), i.e., an arriving customer jumps ahead of the queue and will be served first. We focus on these two strategies in the experiments.

Queue parallelism. It is meant to regulate the capacity of queueing stations when handling multiple requests in parallel. This way, we can consider a number of servers that scales up or down according to the workload that needs to be managed. In general, such a number can be also infinite to model no contention of resources. However, in our case, we are interested in verifying which resources are actually needed to avoid performance issues, and this is the reason why we consider a finite number of servers.

These three elements represent different aspects that are of key relevance for the design of QN models and the performance of the system. The operators in this work target these design elements.

The operators must affect the performance of the model without affecting its semantics, i.e., while the performance of the modelled system should be affected by the mutations introduced by the operators, its functionality must be preserved. This ensures that the mutation analysis process gives insights only on the performance of the model, i.e., by comparing how functionally equivalent models behave under the different workloads. Thus, we need to guarantee that the functional part of the system is not subject to variations, and we do not consider mutation operators that modify the structure of the networks such as adding or removing nodes, rerouting requests, etc. Such mutations could lead, under certain circumstances, to equivalent functionality, for example by merging two parallel nodes that represent the same functionality (e.g., merging the two web servers in Fig. 1 and removing the load balancer). However, performing structural changes to the model without affecting the functionality of the system requires knowledge of the nodes' semantics (the functionality they represent, the potential dependencies between them). Without this information, the mutation could merge two nodes that represent different functionalities, and thus change the semantics of the model. As this knowledge on elements' semantics is not encoded in the model, automating these changes is unfeasible. However, if we could guarantee the preservation of the system's functionality (possibly through semantic annotations in the model), we could envision more mutation operators targeting other queueing network model elements. We leave this aspect as part of our future research.

In this work, we propose the following operators:

Change Queue Size (CQSi): it creates a mutant for each node in the model, modifying the size of its queue. Different versions of this operator decrease or increase the queue size in different ways. $CQSi^{*0.5}$ divides the queue size by 2, and $CQSi^{-1}$ removes one unit; instead, $CQSi^{*2}$ doubles the queue size, and $CQSi^{+1}$ adds one unit. For infinite queues, only $CQSi^{*0.5}$ is applied, and the queue is given an arbitrary finite size (10 in this work).

Change Queue Strategy (CQSt): it creates a mutant for each node in the model, modifying the queueing strategy of its queue. The FCFS and LCFS strategies are swapped, giving the opposite strategy from the original to the mutated queues.

Change Number of Servers (CNS): it creates a mutant for each processing node in the model, modifying its number of servers (processing units). Similarly to *CQSi*, different versions of the operator modify the number in different ways: $CNS^{*0.5}$, CNS^{-1}, CNS^{*2}, and CNS^{+1}.

3.4. Performance metrics

The performance of a particular model *QN* can be assessed using different metrics, that consider different aspects of the performance. This work considers *system response time*, *drop rate*, and *utilisation*, as they represent the metrics most used to assess a system's performance with QNs (Jain, 2008). Other metrics (e.g., *queue length* – the number of requests present in the queue through the simulation – and *throughput*) could be used without otherwise altering the approach.

System response time (*SRT*) is defined as the time interval between a user's request of a service and the response of the system, and usually upper bounds are defined as *business* requirements by stakeholders.

Utilisation (*UT*) is defined as the ratio of busy time of a resource and the total elapsed time of the measurement period. Usually, upper bounds are defined as part of *system* requirements by system engineers on the basis of their experience, or constraints introduced by other concurrent software systems sharing similar hardware characteristics. Note that the utilisation is computed at the *component level*, i.e., there is an utilisation value for each node of the network. We identify with UT_N the utilisation of a given node N.

Drop rate is more specific to the QN formalism, and is defined as the rate at which the customers' requests are dropped from a station or a region because of a constraint, such as a queue reaching its maximum capacity, or the maximum number of customers in a region under analysis. Note that the drop rate can be computed both at the *component level* (i.e., for each node of the network), and at the *system level*. We call the former *drop rate* (*DR*), and we identify with DR_N the drop rate of a given node N; we call the latter *system drop rate* (*SDR*).

A performance metric *PM* can be evaluated with respect to all the customer classes, or for a particular one. We identify with PM^{All} the evaluation for all the customer classes, and with PM^{cc} the evaluation for a given class cc. Note that both types of measurements are valuable, as they can highlight global problems or problems arising for particular types of requests (customer classes).

3.5. Simulation formalism

Given a queueing network *QN* with customer classes $cc_1, \ldots, cc_m$, a *workload* $w = [\lambda_{cc_1}, \ldots, \lambda_{cc_m}]$ is an assignment of service requests, where λ_{cc_i} is the inter-arrival time for customer class cc_i, regulated by a probability distribution.

A queueing network is simulated using a *simulator*. It allows to specify an input workload w, and to select performance metrics of interest; for each performance metric, the designer can also specify the *Maximum Relative Error* ϵ that identifies the precision required in the simulation. It runs until all the performance metrics obtain results in the required maximum relative error ϵ.

We identify with $QN_{PM}(w) = [m_l, m_m, m_u]$ the simulation results for performance metric *PM*, where m_l and m_u are the minimum and maximum values observed for *PM* during the simulation, and m_m is the mean value.

Since the mutants could behave differently from the original network QN, they could take more or less time than QN. Thus, to avoid that the simulation of a mutant QN' takes too much time, we impose a timeout T on its execution.

3.6. Assessing mutant triggering

A necessary (but not sufficient) condition for stating there is a significant change in performance between the original and mutated model is that the difference between the two mean values of PM is greater than a given threshold γ. This threshold should be specified by the designer according to their domain knowledge, considering the precision specified for the simulation. If this is not the case, the simulation results for performance metric PM are considered the same and the evaluation stops. Otherwise, we consider the simulation results different if the two ranges of values produced for PM do not overlap.[1]

We introduce the following predicate to assess whether a mutant has been *triggered* by a workload w for a given performance metric PM

$$\texttt{triggered}(QN, QN', w, PM) = \frac{|m'_m - m_m|}{m_m} > \gamma \wedge (m_u < m'_l \vee m'_u < m_l) \quad (1)$$

where $QN_{PM}(w) = [m_l, m_m, m_u]$ and $QN'_{PM}(w) = [m'_l, m'_m, m'_u]$.

4. Experiments

This section describes the experiments we conducted to evaluate the approach. Replication data, as well as the implementation of the framework are available in the replication package (Laurent et al., 2022).

4.1. Research questions

We identify the following Research Questions (RQs) to assess the approach:

RQ1 (Effectiveness of the mutation operators): Are the proposed operators effective? I.e., do they introduce changes that the workloads detect?

RQ2 (Effect on the different metrics): Which metrics are more affected by the mutation operators? I.e., which are the metrics that more often allow to detect the mutant? Which is the amount of change in the metrics?

RQ3 (Example of insights from the mutation analysis): What kind of conclusions can be drawn by the practitioners?

4.2. Benchmarks and tools used

As tool for QN representation and simulation we use JMT (Casale et al., 2018) ver. 1.1.1; in particular, we use JSIMgraph (the engine part of JMT) to run QN model simulations and collect the performance indices of interest (see Section 3.4).

In order to experiment the approach on real-world case studies, we collected different benchmark QNs from GitHub, that are summarised in Table 1.

The table reports, for each benchmark QN, the original source, the number of workloads used, the number of mutants generated for each mutant operator, and the number of performance metrics for each category. In order to experiment with all our mutation operators, we made the queue sizes finite and increased the number of servers of the original networks, setting both to 10 for all the nodes. The replication package (Laurent et al., 2022) contains the modified models.

Table 1
Benchmark QNs.

ID	Source	$\|W\|$	# mutants			# performance metrics		
			CQSi	*CQSt*	*CNS*	*UT*	*SRT*	*DR/SDR*
B_1	(Comi, 2020), link	3	12	3	12	6	2	8
B_2	(Comi, 2020), link	3	8	2	8	6	3	9
B_3	(ShicaoUW, 2017), link	3	20	5	20	15	3	18
B_4	(ShicaoUW, 2017), link	9	16	4	16	16	4	20
B_5	Arcelli (2020), link	3	36	9	36	189	21	210

Each benchmark QN only had one workload specified by the original developers. Therefore, in order to obtain a fuller picture of the behaviour of the models, and results that would more closely match a real use case (where a system would be simulated under different workloads), for each QN we proceeded as follows. Given a total workload $w = [\lambda_{cc_1}, \ldots, \lambda_{cc_m}]$, for each workload λ_{cc_i} for the customer class cc_i, we multiplied it by 2 and 0.5 and we built two tests $w'_i = [\lambda_{cc_1}, \ldots, \lambda_{cc_i} * 2, \ldots, \lambda_{cc_m}]$ and $w''_i = [\lambda_{cc_1}, \ldots, \lambda_{cc_i} * 0.5, \ldots, \lambda_{cc_m}]$ with the modified values. Moreover, we also build two tests in which we double and halve all the customer classes' loads, i.e., $w' = [\lambda_{cc_1} * 2, \ldots, \lambda_{cc_i} * 2, \ldots, \lambda_{cc_m} * 2]$ and $w'' = [\lambda_{cc_1} * 0.5, \ldots, \lambda_{cc_i} * 0.5, \ldots, \lambda_{cc_m} * 0.5]$. So, in total there are $3 + 2 \times m$ workloads for the network QN, with m the number of customer classes of QN (using an exponential or deterministic distribution.[2] We collected all the workloads in the set W.

4.3. Experimental setup

For each benchmark B_i and its workloads W, an experiment is as follows. B_i is simulated with each w in W and all performance metrics are collected. Then, for each mutant $\mathcal{M}$ of B_i:

- $\mathcal{M}$ is simulated using the workloads in W, and the performance metrics are collected. For a given workload w, the mutant execution is stopped if it times out (after T). In case of timeout, the mutant is considered triggered by w, but no information of the performance metrics can be provided;
- if the timeout for a workload w did not occur, the performance results of B_i and $\mathcal{M}$ are compared as described in Section 3.6.

4.4. Results

This section presents the experimental results, and discusses them in light of the three RQs described in Section 4.1.

4.4.1. RQ1: Effectiveness of the mutation operators

This RQ assesses if the proposed operators can affect system performance. Table 2 reports aggregated experimental results. For each benchmark network and each mutation operator, it reports the proportion of simulations of the mutated networks that were detected as different (DM), i.e., the proportion of times that the predicate in Eq. (1) was true. We observe that, for all mutation operators of type *CNS* (i.e., $CNS^{*0.5}$, CNS^{-1}, CNS^{*2}, and CNS^{+1}), all the obtained mutants are almost always detected. This means that these are mutation operators that can greatly affect (positively or negatively) the performance of the network, highlighting

[1] Note that it could happen that the intervals do not overlap, but the absolute relative change $|m'_m - m_m|/m_m$ is lower than the noise. So, checking overlapping is not enough, and the property on the percentage difference must also be checked.

[2] Note that when there is only one customer class, $w' = w'_1$ and $w'' = w''_1$, and so there are only 3 workloads in total.

Table 2
RQ1: Effectiveness of the mutation operators.

Perf. metrics triggering the mutants (Metr)					Triggered mutants (DM)				
ID	$CNS^{*0.5}$, CNS^{-1}, CNS^{*2}, CNS^{+1}	$CQSi^{*0.5}$	$CQSi^{-1}$	$CQSt$, $CQSi^{+1}$ $CQSi^{*2}$	ID	$CNS^{*0.5}$, CNS^{-1}, CNS^{*2}, CNS^{+1}	$CQSi^{*0.5}$	$CQSi^{-1}$, $CQSi^{+1}$	$CQSt$, $CQSi^{*2}$
B_1	0.12	0	0	0	B_1	1	0	0	0
B_2	0.14	0.17	0	0	B_2	1	0.50	0	0
B_3	0.07	0	0	0	B_3	1	0.40	0	0
B_4	0.10	0.28	0.01	0	B_4	1	0.75	0.03	0
B_5	0.01	0.01	0	0	B_5	0.59	0.07	0	0

Table 3
RQ2 — Minimum and maximum relative change (in %) of each metric created by any mutant under any workload for each system. Grey cells identify the results that actually lead to consider a mutant triggered.

ID	*DR*/*SDR*		*SRT*		*UT*	
	min	max	min	max	min	max
B_1	0	0	0	0	−50	100
B_2	0	+∞	−0.72	0.28	−50	101.74
B_3	0	+∞	−1.09	2.1	−50.22	102.81
B_4	−100	+∞	−6.38	2.37	−51.35	105.82
B_5	−100	+∞	−2.02	1.94	−53.64	123.4

parts of the model that are important to the performance. For $CQSi^{*0.5}$, instead, mutants are detected less often, and, in some cases, they are never detected. This mutation operator does not change the network so drastically, and so it is more difficult to detect. Finally, for some mutation operators (i.e., $CQSi^{-1}$, $CQSi^{+1}$, $CQSt$, and $CQSi^{*2}$), the mutants are (almost) never detected. In this case, the perturbations introduced by the operators are so minimal that they do not significantly impact the performance. Note that a different reason for not detecting a mutant could be that the set of workloads W does not contain a suitable load that triggers the difference. This does not exclude that there might be another workload that would be able to trigger such a difference. In this work, we assume that the typical set of workloads of the system is given (see Section 3.1), and we are not interested in its quality; instead we are concerned about the relative level of difficulty in detecting the mutants across the different types of operators: using the same set of workloads for all the mutants allows to assess this.

Table 2a reports aggregated experimental results regarding the performance metrics. For each benchmark network and each mutation operator, it reports the proportion of metrics that showed a significant difference, i.e., for which the mutated networks were detected. We observe that, in almost all the cases in which at least one mutant is detected (see Table 2b), a small percentage of the metrics is responsible for the detection (up to 28% of the metrics). This means that the mutants result in specific changes to the performance, rather than affecting the performance of the system in all its aspects. The next research question looks more closely at how the different metrics are affected.

4.4.2. RQ2: Effect on the different metrics

This RQ identifies which metrics are more sensitive to the mutations introduced by the mutation operators, i.e., which metrics usually detect the mutants, and which are the changes in their values.

Table 3 reports, for each benchmark and each type of metric *PM*, the minimum and maximum relative change $^{m'_m - m_m}/_{m_m}$ between the average value m_m of *PM* for the original QN and the average value m'_m for a mutant, across all the mutants.

The table reports in grey the results that are, in their absolute value, above the threshold $\gamma = 5\%$ that identifies the percentage change that is necessary in order to assess that a mutant is triggered (see Eq. (1)).

For *DR*/*SDR* and *SRT* in B_1, there is no percentage difference for any mutant, meaning that these metrics were not affected by the mutation. For *DR*/*SDR* in B_2 and B_3, the change is only in the positive direction, meaning that the mutants were only able to increase the value of the performance metric.

We note that the system response time, although almost always slightly affected, is significantly affected only for B_4. This can be explained by the fact that the system response time is a global metric and the effect of local modifications only partially affect the whole system. Indeed, often such local modifications have a high local effect that is, however, masked and compensated at the global level.

On the other hand, the utilisation can always be significantly affected both positively and negatively by some mutants. The reason is that utilisation is a local metric and, as such, it is more sensitive to modifications.

For drop rate and system drop rate, we notice that, for four benchmarks, the maximum value is $+\infty$, meaning that there were no requests dropped in the original model and requests were dropped in the mutated model. This means that the original models have the correct amount of resources (i.e., they are not over-equipped), as modifying them reduces the performance.

4.4.3. RQ3: Example of insights from the mutation analysis

To better illustrate how the proposed approach is able to assist designers in their performance analysis, we provide complete results for one of the benchmark models and discuss the insights they would provide to the system's designer. We select benchmark B_4 (taken from ShicaoUW (2017)), as it is relatively simple and we can present and discuss its results in the space of the paper. The basic idea of this benchmark is to model a source of customers making three types of requests, i.e., Class0, Class1, and Class2, and regulated by different inter-arrival time distributions, that interact with a WebServer. The server processes the different incoming requests and then dispatches the result between three storage entities Storage1, Storage2, and Storage3. Finally the requests reach the sink node.

Tables 4 to 8 report, for each metric *PM* and each mutation operator *MO*, the minimum and maximum difference between the *PM* values of the original network and those of the mutants obtained with *MO*. In the tables, the symbol – means that there is no change in the performance metric. Cells highlighted in grey identify the metric/mutant pairs for which at least one workload triggered the mutant w.r.t. the metric.

These results provide the following insights about the system's performance:

- there exist a correspondence between metrics on a specific node and operators acting on that node. For example, utilisation of Class0 in WebServer is affected by the CNS operator applied on WebServer. More specifically, we can observe that when halving the number of servers, the utilisation is subject to a variation of around 100% (Table 6), i.e., increasing to a large extent. As expected, decreasing the

Table 4
RQ3 – Results for B_4 – *CQSi* reducing.

measureDescr	$CQSi^{*0.5}$Storage1	$CQSi^{*0.5}$Storage2	$CQSi^{*0.5}$Storage3	$CQSi^{*0.5}$WebServer	$CQSi^{-1}$Storage1	$CQSi^{-1}$Storage2	$CQSi^{-1}$Storage3	$CQSi^{-1}$WebServer
Timeout	–	–	–	–	-	-	-	-
$DR^{Class0}_{Storage1}$	0 ; +∞	–	–	–	–	–	–	–
$DR^{Class0}_{Storage2}$	0 ; +∞	+∞; +∞	–	–	–	–	0 ; +∞	0 ; +∞
$DR^{Class0}_{Storage3}$	–	–	0 ; +∞	–	–	–	–	–
$DR^{Class0}_{WebServer}$	0 ; +∞	–	–	+∞; +∞	–	–	0 ; +∞	0 ; +∞
$DR^{Class1}_{Storage1}$	0 ; +∞	–	–	–	–	–	–	–
$DR^{Class1}_{Storage2}$	–	+∞; +∞	–	–	–	0 ; +∞	–	–
$DR^{Class1}_{Storage3}$	–	–	+∞; +∞	–	–	–	–	–
$DR^{Class1}_{WebServer}$	–	–	0 ; +∞	+∞; +∞	–	–	–	0 ; +∞
$DR^{Class2}_{Storage1}$	0 ; +∞	–	–	–	–	–	–	–
$DR^{Class2}_{Storage2}$	–	+∞; +∞	–	–	–	0 ; +∞	–	–
$DR^{Class2}_{Storage3}$	–	–	0 ; +∞	–	–	–	–	–
$DR^{Class2}_{WebServer}$	0 ; 79343.49	−100 ; 0	−28.6 ; 0	43218.46 ; +∞	−100 ; 0	0 ; 2869.8	0 ; 98.53	0 ; 64.33
$DR^{All}_{Storage1}$	0 ; +∞	–	–	–	–	–	–	–
$DR^{All}_{Storage2}$	0 ; +∞	+∞; +∞	–	–	–	0 ; +∞	0 ; +∞	0 ; +∞
$DR^{All}_{Storage3}$	–	–	+∞; +∞	–	–	–	–	–
$DR^{All}_{WebServer}$	0 ; 590.6	−100 ; 0	−35.2 ; 0	63308.79 ; +∞	−100 ; 0	0 ; 1247.65	0 ; 20.12	0 ; +∞
SDR^{Class0}	0 ; +∞	+∞; +∞	0 ; +∞	+∞; +∞	–	–	0 ; +∞	0 ; +∞
SDR^{Class1}	0 ; +∞	+∞; +∞	+∞; +∞	+∞; +∞	–	0 ; +∞	–	0 ; +∞
SDR^{Class2}	0 ; +∞	16849.49 ; +∞	0 ; +∞	43219.2 ; +∞	−100 ; 0	0 ; 65.67	0 ; 98.53	0 ; 64.33
SDR^{All}	0 ; +∞	21085.98 ; +∞	10051.32 ; +∞	54250.37 ; +∞	−100 ; 0	0 ; 23.51	0 ; +∞	0 ; +∞
SRT^{Class0}	−0.59 ; 0.57	−2.6 ; −0.55	−0.75 ; 0.5	−4.56 ; 0.05	−0.97 ; 1.13	−1.08 ; 1.51	−0.82 ; 1.34	−1.35 ; 0.71
SRT^{Class1}	−1.58 ; 1.48	−2.94 ; 1.61	−1.96 ; 1.78	−6.38 ; −0.49	−1.08 ; 1.04	−0.81 ; 1.85	−1.95 ; 1.24	−0.91 ; 0.5
SRT^{Class2}	−0.78 ; 1.28	−2.19 ; 0.65	−0.22 ; 0.9	−3.66 ; 0.86	−0.74 ; 1.03	−0.98 ; 1.43	−1.01 ; 1.35	−0.53 ; 1.07
SRT^{All}	−1.86 ; 0.6	−2.71 ; 0.84	−2.18 ; 0.94	−5.42 ; 0.05	−1.81 ; 1.08	−1.35 ; 1.55	−1.61 ; 1.76	−1.75 ; 1.53
$UT^{Class0}_{Storage1}$	−2.82 ; 0.66	−2.87 ; 0.77	−2.15 ; 1.65	−6.03 ; 0.37	−0.36 ; 1.74	−0.73 ; 1.18	−1.15 ; 1.49	−1.22 ; 1.44
$UT^{Class0}_{Storage2}$	−0.35 ; 0.88	−4.29 ; −1.25	−1.65 ; 1.24	−4.94 ; 0	−0.75 ; 1.81	−1.86 ; 1.92	−1.51 ; 2.45	−1.24 ; 1.65
$UT^{Class0}_{Storage3}$	−0.96 ; 0.56	−1.33 ; −0.43	−1.98 ; 1.89	−4.78 ; 0.24	−1.21 ; 2.03	−1.89 ; 2.57	−0.97 ; 0.73	−2.2 ; 2.02
$UT^{Class0}_{WebServer}$	−2.03 ; −0.07	−2.63 ; 0.28	−0.79 ; 0.52	−4.87 ; −0.08	−1.92 ; 3.03	−1.45 ; 2.67	−2.21 ; 1.35	−1.29 ; 0.37
$UT^{Class1}_{Storage1}$	−0.82 ; 2.8	−3.15 ; −1.37	−2.36 ; 3.97	−6.01 ; 1.64	−0.53 ; 3.01	−1.81 ; 1.39	−1.62 ; 2.45	−1.48 ; 2.59
$UT^{Class1}_{Storage2}$	−2.3 ; 1.12	−3.98 ; 0.63	−2.04 ; 3.07	−8.12 ; 0.62	−2.69 ; 2.82	−0.77 ; 3.59	−2.42 ; 1.96	−1.17 ; 1.65
$UT^{Class1}_{Storage3}$	−2.38 ; 1.34	−2.58 ; −0.32	−3.47 ; 1.99	−7.14 ; 0.29	−1.32 ; 1.75	−5.13 ; 1.47	−1.62 ; 1.59	−1.29 ; 1.5
$UT^{Class1}_{WebServer}$	−1.34 ; 1.34	−3.41 ; 0.42	−2.24 ; 2.24	−7.52 ; 1.02	−1.6 ; 2.84	−3.03 ; 1.45	−2.96 ; 2.7	−1.27 ; 2.25
$UT^{Class2}_{Storage1}$	−1.44 ; 1.1	−1.61 ; 0.58	−0.42 ; 0.84	−3.45 ; 0.56	−1.87 ; 1.71	−0.96 ; 1.56	−2.27 ; 3.6	−0.45 ; 1.53
$UT^{Class2}_{Storage2}$	−1.26 ; 0.52	−4.53 ; 0.64	−0.7 ; 1.3	−6.2 ; 0.81	−3.92 ; 2.38	−1.41 ; 2.85	−2.19 ; 3.51	−0.68 ; 1.66
$UT^{Class2}_{Storage3}$	−2.21 ; 1.39	−1.24 ; 2.12	−0.76 ; 1.99	−4.27 ; 3.42	−2.1 ; 2.22	−1.14 ; 1.46	−1.11 ; 2.26	−0.3 ; 1.88
$UT^{Class2}_{WebServer}$	−1.25 ; 1.64	−1.59 ; 2.22	−1.56 ; 1.57	−5.4 ; 0.39	−1.01 ; 1.45	−1.57 ; 2.17	−1.56 ; 1.58	−0.34 ; 2.67
$UT^{All}_{Storage1}$	−1.23 ; 1.34	−2.76 ; 0.51	−1.3 ; 1.6	−6.6 ; −0.03	−1.37 ; 0.97	−1.76 ; 2.28	−1.19 ; 3	−0.54 ; 1.94
$UT^{All}_{Storage2}$	−2.04 ; 1.97	−5.05 ; 2.23	−2 ; 1.79	−5.9 ; 0.76	−2.04 ; 2.1	−1.48 ; 3.19	−3.64 ; 2.69	−0.47 ; 2.26
$UT^{All}_{Storage3}$	−0.61 ; 0.85	−1.84 ; 1.81	−1.1 ; 1.97	−4.68 ; 0.43	−0.34 ; 1.27	−1.07 ; 2.13	−0.6 ; 2.14	−0.44 ; 1.91
$UT^{All}_{WebServer}$	−2 ; 0.72	−2.11 ; 1.93	−2.2 ; 1.15	−7.89 ; 0.27	−2.07 ; 2.15	−1.92 ; 1.81	−2.02 ; 2.49	−1.88 ; 2.01

number of servers of 1 unit only, instead, has an impact but the utilisation only increased around 11%.

- a similar correspondence is observed when operators are meant to improve the system performance. For example, the utilisation of WebServer decreases by roughly 50% when doubling the number of servers (Table 7), whereas it is observed that augmenting of 1 unit the number of servers brings an improvement of 9% approximately.
- regarding *CQSi* (see Tables 4 and 5), we observe no significant difference for the operators increasing the queue size (Table 5), meaning that the network has enough resources to store incoming requests. It is worth remarking that the operator "increasing the queue size" acts on the *capacity* of the system to queue requests; however, it does not mean that the actual queue length will necessarily increase, it depends on the ability of the system to manage requests in time and avoid queueing. By contrast, the operators halving the queue size (Table 4) lead to significant changes, including requests being dropped, showing that this type of change has a strong impact when applied, lowering the performance, and is thus not a desirable change to make. On the other hand, the operators reducing of one unit the queue size, are (except for one case) never detected, meaning that the network is slightly over-equipped and some queue sizes could be reduced to save system cost, maintaining similar performance results.
- regarding *CNS* (see Tables 6 and 7), we observe that all its operators (increasing and decreasing the number of servers) are detected by the utilisation metric, which is to be expected given the relation between utilisation and number of servers (the workload being processed by more or fewer servers). On the one hand, putting the effects of these changes on the utilisation in relation to those on the drop rate in Table 6 shows that the system does not support these reductions in the number of servers, as requests get dropped. On the other hand, looking at Table 7, it shows that adding servers can reduce the drop rate (since more requests are processed in a lower amount of time, i.e., the number of requests waiting decreases) as

Table 5
RQ3 – Results for B_4 – *CQSi* increasing.

measureDescr	$CQSi^{*2}$_Storage1	$CQSi^{*2}$_Storage2	$CQSi^{*2}$_Storage3	$CQSi^{*2}$_WebServer	$CQSi^{+1}$_Storage1	$CQSi^{+1}$_Storage2	$CQSi^{+1}$_Storage3	$CQSi^{+1}$_WebServer
Timeout	–	–	–	–	–	–	–	–
$DR^{Class0}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class0}_{Storage2}$	–	–	0 ; $+\infty$	–	–	–	0 ; $+\infty$	0 ; $+\infty$
$DR^{Class0}_{Storage3}$	–	–	–	–	–	–	–	–
$DR^{Class0}_{WebServer}$	−82.27 ; 0	−100 ; 0	−23.15 ; 0	−100 ; 0	0 ; 679.92	−100 ; 0	−100 ; 0	−100 ; 0
$DR^{Class1}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class1}_{Storage2}$	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0
$DR^{Class1}_{Storage3}$	–	–	–	–	–	–	–	–
$DR^{Class1}_{WebServer}$	–	–	0 ; $+\infty$	–	–	–	0 ; $+\infty$	–
$DR^{Class2}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class2}_{Storage2}$	−100 ; 0	−100 ; 0	−78.28 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−63.27 ; 0	−100 ; 0
$DR^{Class2}_{Storage3}$	–	–	–	–	–	–	–	–
$DR^{Class2}_{WebServer}$	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	0 ; 83.89	−100 ; 0	−100 ; 0	−100 ; 0
$DR^{All}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{All}_{Storage2}$	−100 ; 0	−100 ; 0	−68.9 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−43.12 ; $+\infty$	−21.86 ; 0
$DR^{All}_{Storage3}$	–	–	–	–	–	–	–	–
$DR^{All}_{WebServer}$	−61.08 ; 0	−100 ; 0	0 ; 124.9	−100 ; 0	0 ; 145.19	−100 ; 0	−23.6 ; 0	−100 ; 0
SDR^{Class0}	−77.43 ; 0	−100 ; 0	−48.92 ; 0	−100 ; 0	0 ; 679.92	−100 ; 0	−63.97 ; $+\infty$	−20.04 ; 0
SDR^{Class1}	−100 ; 0	−100 ; 0	−49.93 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−40.77 ; 0	−100 ; 0
SDR^{Class2}	−100 ; 0	−100 ; 0	−51.08 ; 0	−100 ; 0	0 ; 22.59	−100 ; 0	−17.27 ; 0	−100 ; 0
SDR^{All}	−50.51 ; 0	−100 ; 0	0 ; 40.3	−100 ; 0	0 ; 91.76	−100 ; 0	0 ; $+\infty$	0 ; 37.25
SRT^{Class0}	−1.65 ; 1.49	−1.07 ; 0.88	−1.72 ; 0.78	−1.34 ; 0.15	−1.52 ; 0.9	−1.32 ; 0.77	−1.28 ; 0.99	−2.45 ; 1.5
SRT^{Class1}	−1.13 ; 1.6	−1.65 ; 1.75	−0.91 ; 1.01	−1.58 ; 1.1	−1.21 ; 1.38	−0.76 ; 1.02	−1.15 ; 0.77	−1.26 ; 1.82
SRT^{Class2}	−1.29 ; 1.15	−1.21 ; 2.37	−1.12 ; 1.3	−2.03 ; 2.01	−1.33 ; 0.91	−1.35 ; 1.33	−1.33 ; 1.55	−1.26 ; 1.68
SRT^{All}	−1.37 ; 1.27	−1.01 ; 0.99	−1.31 ; 1.26	−1.74 ; 0.5	−1.42 ; 1.4	−1.19 ; 0.9	−1.47 ; 0.4	−1.16 ; 1.06
$UT^{Class0}_{Storage1}$	−0.94 ; 0.85	−1.3 ; 3.77	−0.92 ; 2.45	−1.35 ; 1.35	−1.39 ; 2.68	−2.28 ; 1.78	−1.92 ; 1.91	−1.82 ; 2.57
$UT^{Class0}_{Storage2}$	−2.86 ; 2.23	−2.7 ; 1.66	−1.16 ; 1.65	−3.56 ; 0.69	−1.99 ; 1.53	−1.94 ; 1.49	−1.33 ; 1.96	−4.9 ; 1.84
$UT^{Class0}_{Storage3}$	−1.37 ; 1.49	−0.94 ; 1.67	−1.9 ; 1.8	−2.45 ; 1.4	−1.41 ; 2.02	−2.06 ; 1.77	−1.17 ; 1.11	−2.82 ; 2.23
$UT^{Class0}_{WebServer}$	−1.12 ; 1.62	−1.08 ; 3.27	−1.42 ; 2.01	−2.03 ; 1.9	−1.22 ; 2.19	−1.5 ; 1.44	−0.97 ; 1.57	−1.55 ; 1.82
$UT^{Class1}_{Storage1}$	−1.15 ; 2.16	−2.26 ; 1.47	−1.52 ; 1.65	−2.06 ; 2.64	−2.54 ; 1.95	−1.97 ; 1.72	−4.39 ; 1.33	−1.98 ; 2.23
$UT^{Class1}_{Storage2}$	−2.79 ; 2.17	−2.91 ; 2.12	−1.88 ; 1.53	−1.9 ; 1.81	−1.78 ; 1.94	−2.49 ; 2.01	−2.6 ; 0.48	−2.94 ; 3.24
$UT^{Class1}_{Storage3}$	−2.05 ; 2.62	−0.77 ; 2.23	−1.43 ; 1.42	−0.9 ; 2.08	−0.62 ; 2.06	−1.06 ; 1.12	−0.8 ; 1.59	−0.9 ; 2.41
$UT^{Class1}_{WebServer}$	−1.38 ; 1.03	−1.36 ; 3.7	−1.11 ; 0.9	−1.56 ; 0.53	−2.73 ; 1.36	−1.56 ; 0.88	−2.77 ; 2.5	−0.88 ; 2.61
$UT^{Class2}_{Storage1}$	−1.53 ; 1.57	−2.25 ; 2.08	−2.23 ; 1.12	−2.42 ; 1.27	−2.59 ; 2.11	−3.84 ; 1.34	−3.08 ; 0.56	−1.43 ; 2.7
$UT^{Class2}_{Storage2}$	−1.19 ; 1.19	−1.23 ; 2.02	−1.1 ; 1.5	−1.24 ; 3.42	−1.35 ; 1.59	−2.01 ; 1.09	−1.43 ; 1.69	−1.1 ; 1.15
$UT^{Class2}_{Storage3}$	−1.99 ; 1.9	−1.27 ; 1.43	−1.71 ; 0.8	−1.82 ; 1.5	−1.26 ; 1.54	−1.12 ; 1.32	−0.98 ; 1.62	−2.69 ; 1.88
$UT^{Class2}_{WebServer}$	−1.67 ; 3.03	−2.13 ; 2.18	−1.19 ; 2.43	−3.78 ; 1.93	−0.69 ; 2.28	−3.38 ; 2.2	−1.11 ; 1.64	−2.42 ; 2.04
$UT^{All}_{Storage1}$	−0.57 ; 2.01	−1.66 ; 2.98	−2.54 ; 2.04	−1.87 ; 1.44	−1.61 ; 1.22	−2.62 ; 1.93	−1.06 ; 1.1	−0.68 ; 1.63
$UT^{All}_{Storage2}$	−1.07 ; 1.29	−0.17 ; 2.1	−1.09 ; 4.39	−1.31 ; 1.29	−1.16 ; 1.49	−2.38 ; 1.34	−2.58 ; 0.62	−1.25 ; 1.59
$UT^{All}_{Storage3}$	−0.78 ; 2.11	−1.39 ; 2.39	−2.45 ; 3.23	−1.52 ; 2.44	−1.3 ; 2.13	−1.38 ; 1.49	−1.37 ; 2.56	−1.18 ; 3.25
$UT^{All}_{WebServer}$	−2.37 ; 2.61	−2.37 ; 1.44	−2.98 ; 1.74	−2.06 ; 1.86	−1.34 ; 2.47	−1.06 ; 1.46	−1.43 ; 2.18	−1.74 ; 2.94

well as the utilisation, and thus could be considered as a refactoring.

- regarding *CQSt* (see Table 8), we note that changing the queueing strategy never leads to significant improvement or deterioration of the performance. This means that the designers can consider changing the queueing strategy, if this leads to other advantages (e.g., lower implementation cost).
- regarding the specific performance metrics, we note that drop rate (both *DR* and *SDR*) is significantly changed only for $CQSi^{*0.5}$ (see Table 4), meaning that only such modification leads to strong drop rate changes. Instead, for other modifications that are detected by utilisation (see Tables 6–7), all the requests are still satisfied and the drop rate is not affected.
- the results also allow to get insights for specific nodes of the network. For example, from Table 4, we observe that halving the queue size for WebServer leads to a significant change of utilisation (for some customer class) of all the other network nodes, of the drop rate for Storage3, of the system drop rate, and of the system response time. This means that WebServer is a central component of the network, and other components depend on it for their performance. Therefore, particular care should be taken when designing it, and deciding its resources.

5. Discussion

This section provides a more general discussion of the proposed approach.

First of all, the approach can be computationally expensive, as it requires generating many mutants and performing long simulations on them. However, the approach can be made scalable in different ways. First of all, the designer can decide to consider only some mutation operators, depending on the type of modifications they are interested in. Moreover, they can also select only some nodes of the network as target of the mutations, namely those that they think are more critical and may require a redesign. Finally, a prioritisation can be imposed over the assessment of mutants, avoiding the execution of some mutants; for example, mutant $CQSi^{-1}$ of a node can be executed before mutant $CQSi^{*0.5}$ and, if mutant $CQSi^{-1}$ is triggered, we can avoid checking mutant

Table 6
RQ3 – Results for B_4 – *CNS* reducing.

measureDescr	$CNS^{*0.5}$_Storage1	$CNS^{*0.5}$_Storage2	$CNS^{*0.5}$_Storage3	$CNS^{*0.5}$_WebServer	CNS^{-1}_Storage1	CNS^{-1}_Storage2	CNS^{-1}_Storage3	CNS^{-1}_WebServer
Timeout	-	-	-	-	-	-	-	-
$DR^{Class0}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class0}_{Storage2}$	–	0 ; $+\infty$	–	–	–	0 ; $+\infty$	–	0 ; $+\infty$
$DR^{Class0}_{Storage3}$	–	0 ; $+\infty$	–	–	–	–	–	–
$DR^{Class0}_{WebServer}$	–	–	0 ; $+\infty$	0 ; $+\infty$	–	–	–	–
$DR^{Class1}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class1}_{Storage2}$	–	–	0 ; $+\infty$	–	0 ; $+\infty$	–	–	–
$DR^{Class1}_{Storage3}$	–	–	0 ; $+\infty$	–	–	–	–	–
$DR^{Class1}_{WebServer}$	–	–	0 ; $+\infty$	0 ; $+\infty$	–	–	0 ; $+\infty$	0 ; $+\infty$
$DR^{Class2}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class2}_{Storage2}$	0 ; $+\infty$	0 ; $+\infty$	–	–	–	–	–	–
$DR^{Class2}_{Storage3}$	–	–	0 ; $+\infty$	–	–	–	–	–
$DR^{Class2}_{WebServer}$	0 ; 85.15	−100 ; 0	−100 ; 0	0 ; $+\infty$	0 ; 81.13	0 ; 149.09	−100 ; 0	−100 ; $+\infty$
$DR^{All}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{All}_{Storage2}$	0 ; $+\infty$	0 ; $+\infty$	0 ; $+\infty$	–	0 ; $+\infty$	0 ; $+\infty$	–	0 ; $+\infty$
$DR^{All}_{Storage3}$	–	0 ; $+\infty$	0 ; $+\infty$	–	–	–	–	–
$DR^{All}_{WebServer}$	−15.98 ; 0	−100 ; 0	−34.07 ; $+\infty$	0 ; $+\infty$	−17.8 ; 0	0 ; 13.03	0 ; 1019.92	−18.6 ; $+\infty$
SDR^{Class0}	–	0 ; $+\infty$	0 ; $+\infty$	0 ; $+\infty$	–	0 ; $+\infty$	–	0 ; $+\infty$
SDR^{Class1}	–	–	0 ; $+\infty$	0 ; $+\infty$	0 ; $+\infty$	–	0 ; $+\infty$	0 ; $+\infty$
SDR^{Class2}	0 ; 192.72	0 ; $+\infty$	0 ; 27.58	0 ; $+\infty$	0 ; 81.13	0 ; 149.09	−100 ; 0	−100 ; $+\infty$
SDR^{All}	0 ; 13.86	0 ; $+\infty$	0 ; $+\infty$	0 ; $+\infty$	0 ; 40.91	0 ; 128.55	0 ; 859.93	0 ; $+\infty$
SRT^{Class0}	−1.23 ; 0.92	−0.43 ; 1.41	−1.96 ; 0.35	−0.72 ; 1.03	−1.44 ; 0.95	−0.86 ; 1.2	−1.19 ; 1.33	−0.88 ; 1.24
SRT^{Class1}	−1.28 ; 1.39	−1.53 ; 0.66	−1.76 ; 1.69	−1.77 ; 0.95	−1.81 ; 0.78	−1.04 ; 1.33	−1.76 ; 1.24	−1.46 ; 0.87
SRT^{Class2}	−1.46 ; 1.66	−1.68 ; 2.06	−0.61 ; 2.03	−1.86 ; 1.38	−1.93 ; 1.98	−0.57 ; 1.06	−0.21 ; 2.16	−0.58 ; 1.43
SRT^{All}	−2.23 ; 0.92	−1.54 ; 1.22	−1.95 ; 1.5	−1.46 ; 1.29	−0.89 ; 1.43	−0.68 ; 1.56	−2.09 ; 0.74	−0.48 ; 1.8
$UT^{Class0}_{Storage1}$	97.56 ; 105.82	−0.81 ; 1.69	−2.09 ; 2.08	−1.49 ; 1.51	9.46 ; 13.16	−2.05 ; 3.8	−1.34 ; 2.73	−1 ; 2.25
$UT^{Class0}_{Storage2}$	−0.65 ; 1.72	97.42 ; 104.81	−2.03 ; 0.92	−0.35 ; 1.83	−1.08 ; 2.27	9.57 ; 13.63	−1.49 ; 1.15	−3.17 ; 1.39
$UT^{Class0}_{Storage3}$	−1.13 ; 1.51	−1.89 ; 1.71	96.76 ; 102.13	−1.35 ; 1.84	−1.95 ; 1.91	−2.57 ; 1.72	8.84 ; 13.58	−2.09 ; 2.04
$UT^{Class0}_{WebServer}$	−1.33 ; 0.95	−1.92 ; 1.09	−2.11 ; 1.15	96.69 ; 103.2	−1.33 ; 1.64	−1.67 ; 1.81	−2.21 ; 1.14	9.32 ; 12.58
$UT^{Class1}_{Storage1}$	96.35 ; 104.13	−1.07 ; 1.99	−1.55 ; 2.12	−0.44 ; 2.08	9.06 ; 13.7	−1.81 ; 1.55	−2.46 ; 2.31	−1.07 ; 1.4
$UT^{Class1}_{Storage2}$	−0.7 ; 3.12	96.72 ; 102.96	−2.12 ; 3.43	−1.58 ; 2.26	−0.79 ; 1.76	8.97 ; 13.32	−4.55 ; 2.83	−1.17 ; 2.71
$UT^{Class1}_{Storage3}$	−2.33 ; 1.19	−2.58 ; 1.89	94.96 ; 103.47	−1.63 ; 1.21	−2.24 ; 0.84	−1.33 ; 1.48	8.84 ; 11.74	−1.9 ; 0.86
$UT^{Class1}_{WebServer}$	−1.95 ; 1.58	−2.17 ; 1.45	−2.33 ; 2.57	95.81 ; 104.09	−1.72 ; 1.78	−2.42 ; 1.84	−2.22 ; 2.94	9.67 ; 12.63
$UT^{Class2}_{Storage1}$	96.72 ; 104.18	−2.58 ; 2.06	−1.99 ; 2.66	−1.66 ; 2.98	9.58 ; 13.44	−1.37 ; 1.53	−1.98 ; 1.94	−0.4 ; 1.97
$UT^{Class2}_{Storage2}$	−1.01 ; 1.86	97.32 ; 105.34	−0.48 ; 1.1	−1.25 ; 2.03	−2.76 ; 2.26	9.89 ; 13.31	−2.6 ; 1.57	−0.59 ; 1.79
$UT^{Class2}_{Storage3}$	−2.87 ; 1.83	−2.68 ; 2.46	99.37 ; 104.42	−1.46 ; 1.38	−2.14 ; 2.59	−2.16 ; 1.78	9.3 ; 13.14	−1.19 ; 1.81
$UT^{Class2}_{WebServer}$	−0.57 ; 1.55	−1.98 ; 2.66	−0.82 ; 2.07	96.36 ; 104.02	−1.17 ; 1.46	−1.27 ; 1.43	−1.66 ; 2.15	9.65 ; 13.55
$UT^{All}_{Storage1}$	97.3 ; 103.93	−1.65 ; 2.76	−1.83 ; 2.44	−2.19 ; 1.74	9.48 ; 12.25	−0.43 ; 2.73	−1.71 ; 1.81	−1.48 ; 2.34
$UT^{All}_{Storage2}$	−2.23 ; 2.3	97.26 ; 103.68	−1.31 ; 2.35	−1.51 ; 2.25	−0.04 ; 1.95	10 ; 14.24	−2.69 ; 1.69	−1.49 ; 3.38
$UT^{All}_{Storage3}$	−0.92 ; 2.35	−1.54 ; 2.95	98.31 ; 105.04	−1.33 ; 2.04	−0.89 ; 1.42	−0.4 ; 2.37	8.92 ; 13.19	−1.51 ; 4.09
$UT^{All}_{WebServer}$	−2.05 ; 1.47	−2.69 ; 1.18	−1.82 ; 2.99	97.23 ; 104.3	−1.29 ; 2.28	−2.63 ; 2.26	−2.92 ; 1.83	9.74 ; 14.09

$CQSI^{*0.5}$, as we know that reducing the queue size has a significant effect.

One may wonder whether the approach is really useful. First of all, the approach reflects a type of analysis that is done, in the current practice, in a manual manner; automatising such an approach is a contribution per se. Moreover, in RQ3, we showed that the approach can indeed lead to some insights that could help in redesigning the network. However, how easy the method is to use has not been assessed. Measuring the *usability* of the method to practitioners would require extensive user studies, which are left as future work.

Note that the domain knowledge of the designer is still important in choosing the alternative design choices. Therefore, although in this paper we experimented with operators using given levels of alteration, these operators can be *adapted* by designers who can decide which alteration levels to use.

6. Threats to validity

The validity of the proposed methodology could be affected by some threats. We discuss them using the classical classification of *construct validity*, *conclusion validity*, *internal validity*, and *external validity* (Wohlin et al., 2012).

Construct validity. A threat of this type is that the metrics we use in the evaluation of the approach do not reflect what we are investigating. First of all, we give a quantitative evaluation of the approach (RQ1 and RQ2): to this aim, we have shown to what extent the different performance metrics are effective in detecting mutants, and, so, if they are really needed in the approach. Moreover, since we are proposing a methodology, we have also done a qualitative evaluation (RQ3) showing which are the benefits for designers in using our approach.

Conclusion validity. A threat of this type is related to the reliability of the measures that are collected (Wohlin et al., 2012). First of all, all the experiments have been run on the same machine, using the same version of JMT. Moreover, the condition for assessing a mutant as triggered (see Eq. (1)) is quite strong, and so it is unlikely that we wrongly deem a mutant as triggered because of errors in measurement.

Another conclusion validity threat could be the selected timeout T for mutant execution. When a mutant times out for a workload, we can not record the performance results it would

Table 7
RQ3 – Results for B_4 – *CNS* increasing.

measureDescr	CNS^{*2}_Storage1	CNS^{*2}_Storage2	CNS^{*2}_Storage3	CNS^{*2}_WebServer	CNS^{+1}_Storage1	CNS^{+1}_Storage2	CNS^{+1}_Storage3	CNS^{+1}_WebServer
Timeout	-	-	-	-	-	-	-	-
$DR^{Class0}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class0}_{Storage2}$	0 ; +∞	0 ; +∞	–	0 ; +∞	0 ; +∞	0 ; +∞	–	–
$DR^{Class0}_{Storage3}$	–	–	–	–	–	–	–	–
$DR^{Class0}_{WebServer}$	−100 ; 0	−18.49 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−63.79 ; +∞
$DR^{Class1}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class1}_{Storage2}$	−100 ; 0	−100 ; 0	−100 ; 0	−64.93 ; 0	−100 ; 0	−100 ; 0	−48.55 ; 0	−100 ; 0
$DR^{Class1}_{Storage3}$	–	–	–	–	–	–	–	–
$DR^{Class1}_{WebServer}$	–	0 ; +∞	–	–	0 ; +∞	–	0 ; +∞	0 ; +∞
$DR^{Class2}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{Class2}_{Storage2}$	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0
$DR^{Class2}_{Storage3}$	0 ; +∞	–	–	–	–	–	–	–
$DR^{Class2}_{WebServer}$	−27.25 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−29.7 ; 0	−100 ; 0	−100 ; 0
$DR^{All}_{Storage1}$	–	–	–	–	–	–	–	–
$DR^{All}_{Storage2}$	0 ; +∞	−56.07 ; 0	−100 ; 0	−64.93 ; 0	−40.52 ; 0	−42.22 ; 0	−74.28 ; 0	−100 ; 0
$DR^{All}_{Storage3}$	0 ; +∞	–	–	–	–	–	–	–
$DR^{All}_{WebServer}$	−51.5 ; 0	0 ; 6.61	−100 ; 0	−100 ; 0	−66.2 ; 0	−53.13 ; 0	0 ; 10.19	0 ; +∞
SDR^{Class0}	0 ; +∞	−39.29 ; 0	−100 ; 0	0 ; 40.55	−39.13 ; 0	−40.88 ; 0	−100 ; 0	−63.79 ; +∞
SDR^{Class1}	−100 ; 0	−51.65 ; 0	−100 ; 0	−64.93 ; 0	−69.9 ; 0	−100 ; 0	0 ; 2.89	−59.47 ; 0
SDR^{Class2}	−3 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−100 ; 0	−53.13 ; 0	−100 ; 0	−100 ; 0
SDR^{All}	0 ; +∞	−1.94 ; 0	−100 ; 0	−38.4 ; 0	−45.1 ; 0	−45.02 ; 0	−9.64 ; 0	−6.78 ; +∞
SRT^{Class0}	−3.54 ; 1.12	−1.28 ; 0.48	−1.65 ; 1.17	−1.48 ; 1.29	−1.66 ; 1.19	−1.54 ; 1.49	−2.32 ; 2.34	−1.67 ; 2.31
SRT^{Class1}	−0.99 ; 1.05	−1.22 ; 0.79	−0.86 ; 1.11	−0.66 ; 1.34	−1.78 ; 1.28	−2.02 ; 0.67	−1.18 ; 0.95	−1.15 ; 1.62
SRT^{Class2}	−1.15 ; 0.68	−1.84 ; 1.78	−0.58 ; 0.52	−1.84 ; 0.91	−1.58 ; 1.03	−1.07 ; 1.22	−1.39 ; 1.25	−1.08 ; 1.54
SRT^{All}	−1.15 ; 1.53	−1.2 ; 1.28	−2.01 ; 0.57	−1.42 ; 1.04	−1.44 ; 0.44	−2.16 ; 0.95	−1.1 ; 0.73	−1.28 ; 0.99
$UT^{Class0}_{Storage1}$	−50.57 ; −49.12	−1.59 ; 1.56	−0.5 ; 1.44	−2.09 ; 2.78	−11.06 ; −6.96	−1.12 ; 2.99	−2.12 ; 3.23	−1.5 ; 1.66
$UT^{Class0}_{Storage2}$	−1.2 ; 1.98	−51.35 ; −49.22	−2.06 ; 2.47	−1.81 ; 1.75	−2.41 ; 0.53	−10.88 ; −7.58	−3.2 ; 1.33	−4.21 ; 2.01
$UT^{Class0}_{Storage3}$	−3.47 ; 1.44	−1.53 ; 1.84	−51.07 ; −48.83	−2.36 ; 1.54	−1.33 ; 1.15	−1.4 ; 1.84	−10.89 ; −6.98	−1.65 ; 2.76
$UT^{Class0}_{WebServer}$	−1.97 ; 2.72	−1.92 ; 1.48	−1.63 ; 3.04	−50.74 ; −48.82	−1.41 ; 1.83	−1.16 ; 1.79	−1.69 ; 2.42	−10.41 ; −8.07
$UT^{Class1}_{Storage1}$	−51.01 ; −49.16	−0.77 ; 0.72	−0.29 ; 2.26	−1.31 ; 1.38	−9.36 ; −7.91	−3.19 ; 3.17	−1.24 ; 2.3	−1.14 ; 1.69
$UT^{Class1}_{Storage2}$	−2.36 ; 1.69	−51.24 ; −49.25	−2.5 ; 2.61	−2.06 ; 1.85	−2.91 ; 1.22	−10.97 ; −7.4	−1.41 ; 1.45	−2.69 ; 1.79
$UT^{Class1}_{Storage3}$	−1.99 ; 1.78	−1.41 ; 1.58	−50.49 ; −49.31	−1.28 ; 1.49	−1.06 ; 1.63	−1.12 ; 2.51	−10.2 ; −7.74	−1.42 ; 1.6
$UT^{Class1}_{WebServer}$	−1.39 ; 1.58	−2.01 ; 0.7	−1.39 ; 1.95	−50.4 ; −48.89	−1.12 ; 1.34	−2.74 ; 1.52	−1.08 ; 1.2	−10.46 ; −8.12
$UT^{Class2}_{Storage1}$	−50.82 ; −49.03	−3.78 ; 1.76	−1.59 ; 0.66	−2.28 ; 0.67	−11.2 ; −7.74	−1.46 ; 1.56	−3.1 ; 1.94	−1.27 ; 2.43
$UT^{Class2}_{Storage2}$	−1.46 ; 0.89	−50.97 ; −49.36	−3.72 ; 1.9	−1.58 ; 0.48	−2.11 ; 2.64	−9.95 ; −7.82	−1.57 ; 2.76	−1.96 ; 1.14
$UT^{Class2}_{Storage3}$	−2.27 ; 1.46	−1.25 ; 1.51	−50.87 ; −48.58	−1.24 ; 1.05	−1.91 ; 1.89	−1.24 ; 0.66	−10.51 ; −7.71	−1.44 ; 1.44
$UT^{Class2}_{WebServer}$	−1.08 ; 2.02	−3.39 ; 2.22	−2.22 ; 1.79	−50.99 ; −49.21	−1.3 ; 2.16	−2.37 ; 2.45	−3.12 ; 1.52	−10.17 ; −6.92
$UT^{All}_{Storage1}$	−50.25 ; −48.44	−4.16 ; 2.19	−1.4 ; 1.28	−3.02 ; 3.61	−10.62 ; −8.11	−0.9 ; 1.68	−1.79 ; 1.71	−0.73 ; 1.72
$UT^{All}_{Storage2}$	−1.76 ; 1.03	−50.8 ; −49.74	−2 ; 1.3	−0.76 ; 1.01	−1.58 ; 1.17	−10.16 ; −8.05	−2.58 ; 1.86	−2.36 ; 1.72
$UT^{All}_{Storage3}$	−1.46 ; 1.73	−1.73 ; 2.63	−51.16 ; −49.24	−2.52 ; 2.17	−1.64 ; 1.4	−1.1 ; 1.53	−10.74 ; −6.76	−0.59 ; 2.14
$UT^{All}_{WebServer}$	−0.94 ; 2.49	−2.11 ; 0.79	−1.52 ; 1.86	−50.78 ; −48.7	−2.23 ; 1.82	−1.02 ; 2.01	−2.33 ; 1.67	−11.19 ; −6.93

have achieved. In this case, the mutant is considered triggered, as the original network did not time out on the same workload, and so it is likely that the simulation result would be different. In any case, to have more confidence that the mutant really affects the performance results, the user should consider to increase the timeout of some workloads.

Internal validity. A threat of this type could be to identify a causal relationship between the workloads and performance metrics used, and the triggered mutants, while the triggering is determined by other factors. For example, a very sensitive detection mechanism could see differences where there are any, just because of the variability of the simulation results: as discussed in Section 6, we defined a restrictive condition on mutant triggering (see Eq. (1)), so as to avoid this problem. As another problem, the variation of the performance results could be due to a wrong implementation of the mutants: to address this problem, we carefully tested our implementation, and we checked that all the produced mutants can be parsed correctly by the JMT tool that we used for simulation.

External validity. A threat of this type is that the approach could not be applicable to other case studies. We acknowledge that we experimented the approach on few benchmark QNs. However, the lack of benchmarks in performance modelling is recognised (Smith et al., 2018; Smith, 2015; Petriu, 2015). Moreover, our approach has been devised for queueing networks, and it is not clear whether it is applicable to other types of performance models (e.g., stochastic Petri Nets, Markov processes, stochastic Process Algebras, etc.). However, all these notations have concepts similar to workload, performance metric, etc.; therefore, as future work we plan to investigate how our approach can be extended and/or adapted for these formalisms.

7. Related work

This section first reviews works related to load assessment in performance analysis (see Section 7.1), that are more closely related to the proposed approach. Then, it reviews works related to mutation analysis, in particular those related to modelling notations and non-functional properties, as the approach takes inspiration from this research area (see Section 7.2). Finally,

Table 8
RQ3 – Results for B_4 – *CQSt*.

measureDescr	*CQSt*_Storage1	*CQSt*_Storage2	*CQSt*_Storage3	*CQSt*_WebServer
Timeout	-	-	-	-
$DR^{Class0}_{Storage1}$	–	–	–	–
$DR^{Class0}_{Storage2}$	–	–	–	0 ; +∞
$DR^{Class0}_{Storage3}$	–	0 ; +∞	–	–
$DR^{Class0}_{WebServer}$	0 ; +∞	0 ; +∞	0 ; +∞	0 ; +∞
$DR^{Class1}_{Storage1}$	–	–	–	–
$DR^{Class1}_{Storage2}$	–	–	0 ; +∞	–
$DR^{Class1}_{Storage3}$	–	–	–	–
$DR^{Class1}_{WebServer}$	–	–	–	0 ; +∞
$DR^{Class2}_{Storage1}$	–	–	–	–
$DR^{Class2}_{Storage2}$	–	–	–	–
$DR^{Class2}_{Storage3}$	–	0 ; +∞	–	–
$DR^{Class2}_{WebServer}$	−100 ; 0	−100 ; 0	−33.44 ; 0	−100 ; 0
$DR^{All}_{Storage1}$	–	–	–	–
$DR^{All}_{Storage2}$	–	–	0 ; +∞	0 ; +∞
$DR^{All}_{Storage3}$	–	0 ; +∞	–	–
$DR^{All}_{WebServer}$	0 ; 10.14	−47.15 ; 0	−24.55 ; 0	−34.01 ; 0
SDR^{Class0}	0 ; +∞	0 ; +∞	0 ; +∞	0 ; +∞
SDR^{Class1}	–	–	0 ; +∞	0 ; +∞
SDR^{Class2}	−100 ; 0	0 ; 94.5	−33.44 ; 0	−100 ; 0
SDR^{All}	−5.6 ; 0	0 ; 35.91	−22.39 ; 0	−24.31 ; 0

measureDescr	*CQSt*_Storage1	*CQSt*_Storage2	*CQSt*_Storage3	*CQSt*_WebServer
Timeout	-	-	-	-
SRT^{Class0}	−0.53 ; 1.19	−1.48 ; 1.22	−1.56 ; 0.84	−0.99 ; 0.59
SRT^{Class1}	−1.77 ; 0.68	−1.31 ; 0.56	−1.64 ; 1.12	−1.63 ; 1.31
SRT^{Class2}	−0.57 ; 1.46	−0.56 ; 1.28	−0.73 ; 0.86	−0.75 ; 2.02
SRT^{All}	−1.94 ; 2.06	−1.35 ; 1.27	−1.27 ; 1.24	−1.71 ; 0.97
$UT^{Class0}_{Storage1}$	−1.36 ; 2.04	−1.49 ; 2.15	−1.41 ; 1.39	−1.27 ; 1.41
$UT^{Class0}_{Storage2}$	−1.73 ; 2.17	−1.84 ; 2.62	−1.19 ; 1.74	−1.57 ; 1.36
$UT^{Class0}_{Storage3}$	−1.55 ; 1.02	−2.71 ; 3.55	−2.14 ; 1.4	−1.98 ; 0.49
$UT^{Class0}_{WebServer}$	−1.79 ; 1.21	−3.22 ; 1.59	−1.18 ; 0.38	−1.1 ; 0.89
$UT^{Class1}_{Storage1}$	−1.58 ; 2.06	−1.71 ; 1.59	−1.82 ; 2.56	−2.1 ; 2.77
$UT^{Class1}_{Storage2}$	−2.57 ; 1.45	−1.39 ; 1.36	−2.16 ; 3.27	−1.07 ; 2.68
$UT^{Class1}_{Storage3}$	−1.13 ; 0.67	−2.98 ; 1.53	−2.51 ; 1.73	−1.42 ; 1.38
$UT^{Class1}_{WebServer}$	−3.89 ; 2.26	−2.59 ; 1.99	−3.74 ; 2.08	−1.82 ; 0.92
$UT^{Class2}_{Storage1}$	−1.43 ; 1.49	−1.08 ; 1.71	−1.3 ; 0.94	−1.08 ; 1.89
$UT^{Class2}_{Storage2}$	−1.06 ; 1.45	−0.99 ; 2.35	−3.04 ; 1.32	−0.88 ; 2.03
$UT^{Class2}_{Storage3}$	−1.3 ; 2	−0.74 ; 2.94	−2.05 ; 1.8	−1.5 ; 2.21
$UT^{Class2}_{WebServer}$	−1.71 ; 1.42	−1.58 ; 1.9	−1.14 ; 1.82	−0.44 ; 2.31
$UT^{All}_{Storage1}$	−0.96 ; 3.65	−1.01 ; 2.6	−0.95 ; 2.36	−1.17 ; 1.88
$UT^{All}_{Storage2}$	−1.51 ; 1.75	−0.85 ; 1.89	−2.27 ; 2.56	−1.52 ; 2.42
$UT^{All}_{Storage3}$	0.28 ; 2.97	−0.64 ; 3.44	−1.81 ; 1.95	−0.82 ; 2.16
$UT^{All}_{WebServer}$	−2.05 ; 3.65	−2.69 ; 1.48	−2.27 ; 1.77	−1.77 ; 2.31

Table 9
Related work: a brief summary.

Approach	*Scope*	*Model*	*#Operators*	*Automated*
∗This paper∗	Mutation-based analysis of performance models	QN	3	✓
Fang et al. (2021), 2021	Parametric model checking on model fragments	MC	–	
Zhao et al. (2020), 2020	Estimation of interval values for Markov chain parameters	MC	2	
Calinescu et al. (2021), 2019	Analysis of QoS characteristics for patterns of service-based systems	MC	3	
Aleti et al. (2018), 2018	Machine learning applied for performance robustness	ML	–	
Calinescu et al. (2018), 2018	Sensitivity analysis with tolerance levels from designers	MC	–	
Incerto et al. (2017), 2017	Performance-based control through parametric queueing networks	QN	3	✓
Su et al. (2016), 2016	Statistical inference methods for the accuracy of results	MC	–	
Filieri et al. (2016), 2016	Self-adaptation driven by non-functional analysis	MC	2	
Moreno et al. (2015), 2015	Latency-aware adaptation under uncertainty	MC	2	

it reviews work around system tuning that, although it shares some similarities with our approach, shows some fundamental differences (see Section 7.3).

7.1. Performance analysis

We here review works related to ours.

Table 9 briefly summarises the most relevant work ordered by year of publication and distinguished by discussing their *scope* along with the indication of the adopted formal *model*, the *number of operators* applied, and if the proposed approach is fully *automated*.

Zhao et al. (2020) use interval Markov chains (MCs), i.e., models with transition probabilities or rates specified as intervals, to verify reliability and performance properties of software systems affected by parametric uncertainty. Two types of changes are applied to MC models, specifically probabilities and rates are adapted through estimators that aim to detect false alarms and missed changes. Calinescu et al. (2021) present an efficient parametric model checking method for the analysis of QoS-based properties, and it is exploited to tune different patterns and their effect on the system parameters under analysis. Three patterns (i.e., sequential, parallel, and probabilistic) for scheduling services are investigated to evaluate their effect on performance and reliability.

In our previous work (Aleti et al., 2018), we investigated robustness focusing on software performance and used polynomial chaos expansion to predict the correlation between uncertain input parameters and performance output results. As opposite to Aleti et al. (2018), in this paper we are interested in identifying which portions of the performance model are more critical than others and whose change in parameters may lead to performance hinders or improvements.

The efficient synthesis of parametric continuous-time Markov chains is tackled in Calinescu et al. (2018) where designers set tolerance levels and sensitivity analysis is executed accordingly. Probabilistic model checking is exploited for multi-objective synthesis to produce Pareto-based solutions.

The focus on the performance characteristics of software systems subject to uncertainties is given in Incerto et al. (2017), where parametric queueing networks (QNs) are used to control the fulfilment of performance requirements at runtime. The problem is encoded through efficient model predictive control, and the solution consists of values assigned to adaptation knobs of three different types: number of servers, routing probabilities, and service rates. QNs translate into Ordinary Differential Equations (ODE) that enjoy an efficient analysis technique, i.e., fluid approximation (Tribastone, 2013). We share the same formalism for performance models, but we do not formulate an optimisation problem, and our operators are aimed at deriving the relationships of model vs performance results' changes.

Su et al. (2016) make use of parametric model checking and investigate its reliability for run-time Quality-of-Service (QoS) evaluation. Two statistical inference methods are proposed and applied to a cloud server management problem.

The verification of QoS-based characteristics is tackled also by Filieri et al. (2016), where temporal properties of Markov models are controlled through a probabilistic model-checking technique. The proposed sensitivity analysis is based on the system parameters that are of two different types, i.e., probabilities and transition rates of Discrete-Time Markov chains (DTMCs).

A proactive approach is pursued by Moreno et al. in Moreno et al. (2015), where there is a look-ahead horizon to look for the adaptation maximizing the expected utility and taking into account the uncertainty of the environment. Two types of adaptation tactics have been defined, specifically: (i) (ii) change the number of servers, and (iii) modify the dimmer, i.e., the service time of handling requests.

7.2. Mutation analysis

Mutation analysis has been originally proposed for assessing the quality of test suites in program testing (DeMillo et al., 1978), and it has become a mature field, with several tools (Kintis et al., 2018), and industrial adoption (Petrović and Ivanković, 2018; Petrović et al., 2021; Beller et al., 2021). In mutation analysis, the quality of a test suite is determined by the percentage of mutants that are detected (*killed* in the mutation analysis terminology), i.e., that produce results different from the expected ones. Please refer to Papadakis et al. (2019) for an extensive survey.

In our approach, we apply mutation analysis to models. In the literature, different works have proposed mutation operators and corresponding mutation analysis approaches for models, such as Finite State Machines (Fabbri et al., 1994), Circus models (Alberto et al., 2017), Z specifications (Aydal et al., 2009), Estelle specifications (De Souza et al., 1999), Petri nets (Fabbri et al., 1995), Simulink models (Matinnejad et al., 2016), models for model checkers (Arcaini et al., 2015a; Lee and Hsiung, 2004), UML models (Krenn et al., 2015), algebraic specifications (Woodward, 1993), feature models (Arcaini et al., 2015b), and regular expressions (Arcaini et al., 2019). All the previous approaches apply mutation to functional specifications, and the applied mutations have the aim of affecting the system's behaviour.

Few approaches have been proposed for mutation analysis of non-functional properties, as we do in our work. Most of these non-functional mutation works focus on mutation of source code. For example, Lisper et al. (2017) introduced the concept of *targeted mutation*, which consists in mutating the part or the code that is more likely to affect the non-functional property under study, worst-case execution time in their case. Some approaches have been proposed to target specific non-functional properties for code. For example, Loise et al. (2017) proposed to use mutation analysis for security, and they introduced 15 security-aware mutation operators for Java code, showing that they are more effective in testing security aspects of applications.

The application of mutations to models for non-functional testing (as we do in our work), has already been proposed in the past, in particular for timed automata modelling real-time systems. Aichernig et al. (2013) proposed a mutation testing approach for timed automata; they provided different mutation operators for timed automata, and a test case generation approach that generates tests able to kill the mutants. While most of the proposed mutation operators are functional ones (e.g., *change action* or *change guard*), others are more related to the temporal execution of the automaton (e.g., *change invariant*). AbouTrab et al. (2013) tested real-time embedded systems (namely a production cell system) in a conformance testing setting using timed automata: first, mutations resembling possible faults of the corresponding C implementation are injected into the timed automaton acting as specification; then, these mutated timed automata are used to measure the fault detection capability of the test suite. Ortiz Vega et al. (2018) provided a survey on mutation testing for timed systems, those using explicit clocks, such as timed automata, and those using implicit clocks, such as Simulink and Lustre. Starting from the surveyed works, they presented a research agenda where they identified, as possible future research directions, (i) model-in-the-loop to reduce the simulation effort, (ii) more research on the definition of mutation operators for implicit clock models, and (iii) research on the detection of equivalent mutants. Detection of equivalent mutants is also relevant to the context of this work, as an equivalent mutant constitutes a possible valid refactoring of the model.

Our work has some similarities with *performance mutation testing* (PMT) (Delgado-Pérez et al., 2021). Similarly to classical mutation testing, PMT mutates the program under test with some mutation operators that produce functional-equivalent mutants that, however, have a lower performance. Such mutants are used to assess the quality of the test suite for the original system. Our context is different from PMT, as we assume that the provided test suite has a good quality (i.e., the set of workloads characterises the typical load of the system), and we use mutations to discover bottlenecks or over-equipment of the system, and identify possible refactoring actions. However, our approach could be easily adapted to perform model-based PMT, and we leave this as future work. Indeed, the theoretical setting as described in Delgado-Pérez et al. (2021) is similar to ours: mutants must be functionally equivalent, and a mutant is considered killed if it affects the performance in a significant way.

Temple et al. (2021) start from the observation that defining the killing of a performance mutant using absolute numbers is not too significant, as this does not consider the difficulty on the task. Therefore, they propose *multimorphic testing* in which the quality of a test suite is given in terms of its ability to discriminate among different mutants (called *morphs* in their work).

In classic mutation testing for functional testing, *equivalent mutants* are a problem: since they cannot be killed by any test, they reduce the mutation score for no reason. Therefore, different techniques have been proposed to identify equivalent mutants (Papadakis et al., 2015; Schuler and Zeller, 2013; Offutt and Pan, 1997; Madeyski et al., 2014), so as to remove them from the set of considered mutants; note that the problem is, in general, undecidable (Budd and Angluin, 1982). However, recently some works advocate the use of equivalent mutants to improve the "quality" of code and models (Arcaini et al., 2016), to remove *static anomalies* to improve qualities like readability, compactness, efficiency an so on. Such an idea has been pursed by López et al. (2018) who proposed to use equivalent mutants for improving source code optimisation. Our work also goes in the direction advocated in Arcaini et al. (2016): indeed, the mutation operators we propose generate, by definition, functionally equivalent mutants, with the aim of improving the system's performance.

7.3. System tuning

The proposed approach has similarities with approaches that aim to find the best configuration for configurable optimisation algorithms, whose performance depends on the setting of their input parameters. For example, López-Ibáñez et al. (2016) propose *irace*, an approach that iteratively samples configurations of the algorithm, and compares them using different instances of the problem to be optimised; comparisons are performed using statistical tests.

The main differences of *irace* with ours are as follows. *irace* focuses on the improvement of the system and so it discards changes that worsen the performance; in our case, instead, alternative configurations that are worse are still shown to the designers, so that they can understand which nodes are important

and not over/under-equipped; in this sense, our approach aims also at providing some "explanation" of the system performance. Still in this direction, our approach, differently from *irace*, only applies single changes, as the designer can better understand the impact of a single change rather than multiple simultaneous changes. This way, designers are more confident in applying a refactoring action suggested by a change that improves the performance. As a minor note, approaches such as *irace* tend to assess a much larger number of configurations (hence more computationally expensive) than those assessed in our approach; however, we acknowledge that they could also be executed using a number of configurations similar to that used in our approach.

8. Conclusion

Queueing networks have been successfully applied in the past for performance analysis. This type of evaluation usually aims at checking whether the system meets performance requirements, and whether the cost of building the system is justified, i.e., to understand if the system is over-equipped or not. To this end, a system designer usually tries different model versions by adding or removing resources from the model to check whether performance results are affected. This work automatises such a manual approach, by means of a mutation-based framework. We propose a set of mutation operators that reflect the modifications usually performed by the designers, and define a technique to check whether the simulation results of the original model and of a mutant are different. The framework automatically evaluates the different mutants, and provides information on how the mutants impact the different performance metrics of interest; such information can be analysed by the designer, who can eventually decide how to modify the system based on the mutants' impact on the performance.

It is worth remarking that we consider "first order mutation" in this paper. Running combinations (i.e., higher order mutation) would result in a combinatorial number of mutants which might be too expensive, and possibly preventing the adoption of the technique itself. The interaction among operators, although an interesting problem, is out of scope for this work. Our method highlights interesting queueing network changes that the designer can then manually combine and test. Furthermore, a founding principle of mutation analysis, i.e., the coupling effect (Offutt, 1989, 1992), suggests that the insights from first order mutants could subsume those of the higher order mutants, although verifying that this principle holds when mutating queueing networks remains as future work.

The set of proposed mutation operators is aimed to trigger changes that, by construction, preserve the system functionality, as this is required for the performance-based comparison. However, as future work we plan to explore the possibility of introducing ad-hoc annotations in queueing network models that may lead to the specification of new mutation operators targeting pre-defined model elements. As further investigation, we need to study the required effort of annotating queueing network models and the impact of subsequent allowed changes, in order to understand the benefit of this future research direction.

This work considered queueing networks as performance model notation. However, other notations such as stochastic Petri Nets, Markov processes, stochastic Process Algebras, could be used for performance modelling; exploring the applicability of the proposed approach to those formalisms is future work.

The approach has been experimented on a set of QNs from public repositories. In order to draw more definitive conclusions, a larger set of benchmarks would be required, which, however, is missing in the research community. One solution could be to rely on synthetic benchmarks that, however, would only allow to assess quantitative aspects such as the scalability of the approach, but not how meaningful the insights from the method are (as the synthetic networks would be meaningless). A different approach could be to translate, when possible, models developed in other performance notations to QNs; we plan to explore such direction as future work.

As in any model-based approach, the fidelity of the model determines the accuracy of the performance predictions w.r.t. the real system; as such, model fidelity is not a limitation intrinsic to our approach. However, our approach suggests modifications that, at the model level, show some advantage (e.g., reduced cost), and that should be later reflected in the implementation; it is not clear whether these modifications preserve model fidelity, or they introduce some noise. As future work, we plan to integrate a real target implementation in the approach to assess to what extent the benefit estimated by the modification on the model is reflected in a real performance improvement on the actual implementation.

CRediT authorship contribution statement

Thomas Laurent: Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft. **Paolo Arcaini:** Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft. **Catia Trubiani:** Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft. **Anthony Ventresque:** Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We would like to thank the Editor and the anonymous reviewers for their valuable feedback. This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094_P2. P. Arcaini is supported by Engineerable AI Project (No. JPMJMI20B8), Japan Science and Technology Agency; and ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), Japan Science and Technology Agency, Funding Reference number: 10.13039/501100009024 ERATO. This work has been partially funded by MIUR project PRIN 2017TWRCNB SEDUCE (Designing Spatially Distributed Cyber-Physical Systems under Uncertainty).

References

AbouTrab, M.S., Brockway, M., Counsell, S., Hierons, R.M., 2013. Testing real-time embedded systems using timed automata based approaches. J. Syst. Softw. 86 (5), 1209–1223. http://dx.doi.org/10.1016/j.jss.2012.12.030.

Aichernig, B.K., Lorber, F., Ničković, D., 2013. Time for Mutants — Model-based mutation testing with timed automata. In: Tests and Proofs. Springer Berlin Heidelberg, pp. 20–38. http://dx.doi.org/10.1007/978-3-642-38916-0_2.

Alberto, A., Cavalcanti, A., Gaudel, M.-C., Simão, A., 2017. Formal mutation testing for Circus. Inf. Softw. Technol. 81, 131–153. http://dx.doi.org/10.1016/j.infsof.2016.04.003.

Aleti, A., Trubiani, C., van Hoorn, A., Jamshidi, P., 2018. An efficient method for uncertainty propagation in robust software performance estimation. J. Syst. Softw. 138, 222–235. http://dx.doi.org/10.1016/j.jss.2018.01.010.

Antonelli, F., Cortellessa, V., Gribaudo, M., Pinciroli, R., Trivedi, K.S., Trubiani, C., 2020. Analytical modeling of performance indices under epistemic uncertainty applied to cloud computing systems. Future Gener. Comput. Syst. 102, 746–761. http://dx.doi.org/10.1016/j.future.2019.09.006.

Arcaini, P., Gargantini, A., Riccobene, E., 2015a. Using mutation to assess fault detection capability of model review. Softw. Test. Verif. Reliab. 25 (5–7), 629–652. http://dx.doi.org/10.1002/stvr.1530.

Arcaini, P., Gargantini, A., Riccobene, E., 2019. Fault-based test generation for regular expressions by mutation. Softw. Test. Verif. Reliab. 29 (1–2), e1664. http://dx.doi.org/10.1002/stvr.1664.

Arcaini, P., Gargantini, A., Riccobene, E., Vavassori, P., 2016. A novel use of equivalent mutants for static anomaly detection in software artifacts. Inf. Softw. Technol. 81, 52–64. http://dx.doi.org/10.1016/j.infsof.2016.01.019.

Arcaini, P., Gargantini, A., Vavassori, P., 2015b. Generating tests for detecting faults in feature models. In: Proceedings of the International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 1–10. http://dx.doi.org/10.1109/ICST.2015.7102591.

Arcelli, D., 2020. A multi-objective performance optimization approach for self-adaptive architectures. In: Proceedings of European Conference on Software Architecture. ECSA, Springer, Cham, pp. 139–147. http://dx.doi.org/10.1007/978-3-030-58923-3_9.

Aydal, E.G., Paige, R.F., Utting, M., Woodcock, J., 2009. Putting formal specifications under the magnifying glass: Model-based testing for validation. In: Proceedings of the International Conference on Software Testing Verification and Validation. ICST, IEEE Computer Society, USA, pp. 131–140. http://dx.doi.org/10.1109/ICST.2009.20.

Balsamo, S., Marzolla, M., 2005. Performance evaluation of UML software architectures with multiclass queueing network models. In: Proc. of the 5th International Workshop on Software and Performance. In: WOSP '05, ACM, New York, NY, USA, pp. 37–42. http://dx.doi.org/10.1145/1071021.1071025.

Beller, M., Wong, C.-P., Bader, J., Scott, A., Machalica, M., Chandra, S., Meijer, E., 2021. What it would take to use mutation testing in industry-a study at Facebook. In: Proceedings of the IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP, pp. 268–277. http://dx.doi.org/10.1109/ICSE-SEIP52600.2021.00036.

Bondi, A., 2014. Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice. Addison Wesley.

Budd, T.A., Angluin, D., 1982. Two notions of correctness and their relation to testing. Acta Inform. 18 (1), 31–45. http://dx.doi.org/10.1007/BF00625279.

Calinescu, R., Češka, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N., 2018. Efficient synthesis of robust models for stochastic systems. J. Syst. Softw. 143, 140–158. http://dx.doi.org/10.1016/j.jss.2018.05.013.

Calinescu, R., Paterson, C., Johnson, K., 2021. Efficient parametric model checking using domain knowledge. IEEE Trans. Softw. Eng. 47 (6), 1114–1133. http://dx.doi.org/10.1109/TSE.2019.2912958.

Casale, G., Personé, V.D.N., Smirni, E., 2016. QRF: An optimization-based framework for evaluating complex stochastic networks. ACM Trans. Model. Comput. Simul. 26 (3), http://dx.doi.org/10.1145/2724709.

Casale, G., Serazzi, G., Zhu, L., 2018. Performance evaluation with java modelling tools: A hands-on introduction. SIGMETRICS Perform. Eval. Rev. 45 (3), 246–247. http://dx.doi.org/10.1145/3199524.3199567.

Comi, D., 2020. Exercises for course on "computing infrastructures". https://github.com/comidan/Computer-Science-Engineering/tree/master/Master%20Level%20Degree/Computing%20Infrastructures/Lectures/JMT-Examples.

Cortellessa, V., Di Marco, A., Inverardi, P., 2011. Model-Based Software Performance Analysis. Springer.

De Souza, S.D.R.S., Maldonado, J.C., Fabbri, S.C.P.F., De Souza, W.L., 1999. Mutation testing applied to estelle specifications. Softw. Qual. Control 8, 285–301. http://dx.doi.org/10.1023/A:1008978021407.

Delgado-Pérez, P., Sánchez, A.B., Segura, S., Medina-Bulo, I., 2021. Performance mutation testing. Softw. Test. Verif. Reliab. 31 (5), e1728. http://dx.doi.org/10.1002/stvr.1728, e1728 stvr.1728.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: Help for the practicing programmer. Computer 11 (4), 34–41. http://dx.doi.org/10.1109/C-M.1978.218136.

Dipietro, S., Casale, G., Serazzi, G., 2016. A queueing network model for performance prediction of apache cassandra. In: Proceedings of the International Conference on Performance Evaluation Methodologies and Tools. VALUETOOLS, http://dx.doi.org/10.4108/eai.25-10-2016.2266606.

Fabbri, S.C.P.F., Delamaro, M.E., Maldonado, J.C., Masiero, P.C., 1994. Mutation analysis testing for finite state machines. In: Proceedings of International Symposium on Software Reliability Engineering. pp. 220–229. http://dx.doi.org/10.1109/ISSRE.1994.341378.

Fabbri, S.C.P.F., Maldonado, J.C., Masiero, P.C., Delamaro, M.E., Wong, E., 1995. Mutation testing applied to validate specifications based on Petri nets. In: Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII. Chapman & Hall, Ltd., GBR, pp. 329–337. http://dx.doi.org/10.1007/978-0-387-34945-9_24.

Fang, X., Calinescu, R., Gerasimou, S., Alhwikem, F., 2021. Fast parametric model checking through model fragmentation. In: Proceedings of the IEEE/ACM International Conference on Software Engineering. ICSE, pp. 835–846. http://dx.doi.org/10.1109/ICSE43902.2021.00081.

Filieri, A., Tamburrelli, G., Ghezzi, C., 2016. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. IEEE Trans. Softw. Eng. 42 (1), 75–99. http://dx.doi.org/10.1109/TSE.2015.2421318.

Gazzola, L., Micucci, D., Mariani, L., 2019. Automatic software repair: A survey. IEEE Trans. Softw. Eng. 45 (1), 34–67. http://dx.doi.org/10.1109/TSE.2017.2755013.

Harman, M., O'Hearn, P., 2018. From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In: Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 1–23. http://dx.doi.org/10.1109/SCAM.2018.00009.

Haskins, B., Stecklein, J., Dick, B., Moroney, G., Lovell, R., Dabney, J., 2004. Error cost escalation through the project life cycle. In: INCOSE International Symposium, Vol. 14. (1), pp. 1723–1737. http://dx.doi.org/10.1002/j.2334-5837.2004.tb00608.x.

Incerto, E., Napolitano, A., Tribastone, M., 2021. Learning queuing networks via linear optimization. In: Proceedings of the International Conference on Performance Engineering. ICPE, pp. 51–60. http://dx.doi.org/10.1145/3427921.3450245.

Incerto, E., Tribastone, M., Trubiani, C., 2017. Software performance self-adaptation through efficient model predictive control. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. In: ASE 2017, IEEE Press, pp. 485–496. http://dx.doi.org/10.1109/ASE.2017.8115660.

Jain, R., 2008. The Art of Computer Systems Performance Analysis. John Wiley & Sons.

Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., Le Traon, Y., 2018. How effective are mutation testing tools? An empirical analysis of java mutation testing tools with manual analysis and real faults. Empirical Softw. Engg. 23 (4), 2426–2463. http://dx.doi.org/10.1007/s10664-017-9582-5.

Kleinrock, L., 1975. Theory, Volume 1, Queueing Systems. Wiley-Interscience, New York, NY, USA.

Koziolek, H., 2010. Performance evaluation of component-based software systems: A survey. Perform. Eval. 67 (8), 634–658. http://dx.doi.org/10.1016/j.peva.2009.07.007.

Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jobstl, E., Brandl, H., 2015. MoMut::UML model-based mutation testing for UML. In: IEEE International Conference on Software Testing, Verification and Validation. ICST, IEEE Computer Society, Los Alamitos, CA, USA, pp. 1–8. http://dx.doi.org/10.1109/ICST.2015.7102627.

Laurent, T., Arcaini, P., Trubiani, C., Ventresque, A., 2022. Replication package for the paper "Mutation-based Analysis of Queueing Network Performance Models". Zenodo, http://dx.doi.org/10.5281/zenodo.6545506.

Lazowska, E.D., Zahorjan, J., Scott Graham, G., Sevcik, K.C., 1984. Computer system analysis using queueing network models. Prentice-Hall, Inc., Englewood Cliffs.

Lee, T.-C., Hsiung, P.-A., 2004. Mutation coverage estimation for model checking. In: Automated Technology for Verification and Analysis. In: Lecture Notes in Computer Science, vol. 3299, Springer, pp. 354–368. http://dx.doi.org/10.1007/978-3-540-30476-0_29.

Lisper, B., Lindström, B., Potena, P., Saadatmand, M., Bohlin, M., 2017. Targeted mutation: Efficient mutation analysis for testing non-functional properties. In: IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, pp. 65–68. http://dx.doi.org/10.1109/ICSTW.2017.18.

Loise, T., Devroey, X., Perrouin, G., Papadakis, M., Heymans, P., 2017. Towards security-aware mutation testing. In: IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, pp. 97–102. http://dx.doi.org/10.1109/ICSTW.2017.24.

López, J., Kushik, N., Yevtushenko, N., 2018. Source code optimization using equivalent mutants. Inf. Softw. Technol. 103, 138–141, https://doi.org/10.1016/j.infsof.2018.06.013.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., Stützle, T., 2016. The irace package: Iterated racing for automatic algorithm configuration. Oper. Res. Perspect. 3, 43–58. http://dx.doi.org/10.1016/j.orp.2016.09.002.

Madeyski, L., Orzeszyna, W., Torkar, R., Józala, M., 2014. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Softw. Eng. 40 (1), 23–42. http://dx.doi.org/10.1109/TSE.2013.44.

Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T., 2016. Automated test suite generation for time-continuous Simulink models. In: Proceedings of the International Conference on Software Engineering. ICSE, In: ICSE '16, ACM, New York, NY, USA, pp. 595–606. http://dx.doi.org/10.1145/2884781.2884797.

Mishra, K., Trivedi, K.S., 2011. Uncertainty propagation through software dependability models. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering. In: ISSRE '11, IEEE Computer Society, USA, pp. 80–89. http://dx.doi.org/10.1109/ISSRE.2011.14.

Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B., 2015. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In: Proceedings of the Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2015, ACM, New York, NY, USA, pp. 1–12. http://dx.doi.org/10.1145/2786805.2786853.

Offutt, A.J., 1989. The coupling effect: Fact or fiction. SIGSOFT Softw. Eng. Notes 14 (8), 131–140. http://dx.doi.org/10.1145/75309.75324.

Offutt, A.J., 1992. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol. (TOSEM) 1 (1), 5–20. http://dx.doi.org/10.1145/125489.125473.

Offutt, A.J., Pan, J., 1997. Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verif. Reliab. 7 (3), 165–192. http://dx.doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U.

Ortiz Vega, J.J., Perrouin, G., Amrani, M., Schobbens, P.-Y., 2018. Model-based mutation operators for timed systems: A taxonomy and research agenda. In: 2018 IEEE International Conference on Software Quality, Reliability and Security. QRS, pp. 325–332. http://dx.doi.org/10.1109/QRS.2018.00045.

Papadakis, M., Jia, Y., Harman, M., Le Traon, Y., 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: Proceedings of IEEE/ACM International Conference on Software Engineering, Vol. 1. ICSE, pp. 936–946. http://dx.doi.org/10.1109/ICSE.2015.103.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2019. Chapter six - mutation testing advances: An analysis and survey. In: Memon, A.M. (Ed.), In: Advances in Computers, vol. 112, Elsevier, pp. 275–378. http://dx.doi.org/10.1016/bs.adcom.2018.03.015.

Petriu, D.C., 2015. Challenges in integrating the analysis of multiple non-functional properties in model-driven software engineering. In: Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development. In: WOSP '15, ACM, New York, NY, USA, pp. 41–46. http://dx.doi.org/10.1145/2693561.2693566.

Petriu, D.C., 2021. Integrating the analysis of multiple non-functional properties in model-driven engineering. Softw. Syst. Model. 20 (6), 1777–1791. http://dx.doi.org/10.1007/s10270-021-00953-3.

Petriu, D.C., Alhaj, M., Tawhid, R., 2012. Software performance modeling. In: Formal Methods for Model-Driven Engineering: International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM). Advanced Lectures. Springer Berlin Heidelberg, pp. 219–262. http://dx.doi.org/10.1007/978-3-642-30982-3_7.

Petrović, G., Ivanković, M., 2018. State of mutation testing at Google. In: Proceedings of the International Conference on Software Engineering: Software Engineering in Practice. In: ICSE-SEIP '18, ACM, New York, NY, USA, pp. 163–171. http://dx.doi.org/10.1145/3183519.3183521.

Petrović, G., Ivanković, M., Fraser, G., Just, R., 2021. Does mutation testing improve testing practices? In: Proceedings of the IEEE/ACM International Conference on Software Engineering. ICSE, pp. 910–921. http://dx.doi.org/10.1109/ICSE43902.2021.00087.

Pinciroli, R., Trivedi, K., Bobbio, A., 2017. Parametric sensitivity and uncertainty propagation in dependability models. In: Proceedings of the EAI International Conference on Performance Evaluation Methodologies and Tools. In: VALUETOOLS, pp. 44–51. http://dx.doi.org/10.4108/eai.25-10-2016.2266529.

Schuler, D., Zeller, A., 2013. Covering and uncovering equivalent mutants. Softw. Test. Verif. Reliab. 23 (5), 353–374. http://dx.doi.org/10.1002/stvr.1473.

ShicaoUW, 2017. Examples for "QN-ACTR cognitive architecture". https://github.com/HOMlab/QN-ACTR-Release/tree/master/QN-ACTR%20Java/examples.

Smith, C.U., 2015. Software performance engineering then and now: A position paper. In: Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development. In: WOSP '15, ACM, New York, NY, USA, pp. 1–3. http://dx.doi.org/10.1145/2693561.2693567.

Smith, C.U., Cortellessa, V., Gómez, A., Kounev, S., Lladó, C., Woodside, M., 2018. Challenges in automating performance tool support. In: Companion of the ACM/SPEC International Conference on Performance Engineering. ICPE, In: ICPE '18, ACM, New York, NY, USA, pp. 175–176. http://dx.doi.org/10.1145/3185768.3186410.

Smith, C.U., Lladó, C.M., Puigjaner, R., 2010. Performance model interchange format (PMIF 2): A comprehensive approach to queueing network model interoperability. Perform. Eval. 67 (7), 548–568. http://dx.doi.org/10.1016/j.peva.2010.01.006.

Smith, C.U., Williams, L.G., 2002. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley.

Su, G., Rosenblum, D.S., Tamburrelli, G., 2016. Reliability of run-time quality-of-service evaluation using parametric model checking. In: Proceedings of the International Conference on Software Engineering. In: ICSE '16, ACM, New York, NY, USA, pp. 73–84. http://dx.doi.org/10.1145/2884781.2884814.

Temple, P., Acher, M., Jézéquel, J.-M., 2021. Empirical assessment of multimorphic testing. IEEE Trans. Softw. Eng. 47 (7), 1511–1527. http://dx.doi.org/10.1109/TSE.2019.2926971.

Tribastone, M., 2013. A fluid model for layered queueing networks. IEEE Trans. Softw. Eng. 39 (6), 744–756. http://dx.doi.org/10.1109/TSE.2012.66.

Trubiani, C., Meedeniya, I., Cortellessa, V., Aleti, A., Grunske, L., 2013. Model-based performance analysis of software architectures under uncertainty. In: Proceedings of the International Conference on Quality of Software Architectures. ACM, New York, NY, USA, pp. 69–78. http://dx.doi.org/10.1145/2465478.2465487.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A., 2005. An analytical model for multi-tier internet services and its applications. In: Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. In: SIGMETRICS '05, ACM, New York, NY, USA, pp. 291–302. http://dx.doi.org/10.1145/1064212.1064252.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A., 2012. Experimentation in Software Engineering. Springer, http://dx.doi.org/10.1007/978-3-642-29044-2.

Woodward, M.R., 1993. Errors in algebraic specifications and an experimental mutation testing tool. Softw. Eng. J. 8 (4), 211–224. http://dx.doi.org/10.1049/sej.1993.0027.

Xu, J., 2012. Rule-based automatic software performance diagnosis and improvement. Perform. Eval. 69 (11), 525–550. http://dx.doi.org/10.1016/j.peva.2009.11.003.

Zhao, X., Calinescu, R., Gerasimou, S., Robu, V., Flynn, D., 2020. Interval change-point detection for runtime probabilistic model checking. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. In: ASE '20, ACM, New York, NY, USA, pp. 163–174. http://dx.doi.org/10.1145/3324884.3416565.

Thomas Laurent is a postdoctoral research fellow at University College Dublin, Ireland. His main research interests are test criteria and mutation analysis.

Paolo Arcaini is project associate professor at the National Institute of Informatics (NII), Japan. His main research interests are related to search-based testing, fault-based testing, software product lines, and automatic repair. His home page is https://group-mmm.org/~arcaini/.

Catia Trubiani is assistant professor at the Gran Sasso Science Institute (GSSI), Italy. Her main research interests are related to. performance modelling and analysis of interacting heterogeneous distributed systems. Her home page is https://cs.gssi.it/catia.trubiani/.

Dr Anthony Ventresque founded and leads the UCD Complex Software Lab. Dr Ventresque received his Ph.D. degree in Computer Science from the University of Nantes and INRIA France in 2008. He is currently a Lecturer in the School of Computer Science at University College Dublin, Ireland, and a Funded Investigator with Lero, the SFI Irish Software Research Centre.