QN-based Modeling and Analysis of Software Performance Antipatterns for Cyber-Physical Systems

Riccardo Pinciroli
Gran Sasso Science Institute
L'Aquila, Italy
riccardo.pinciroli@gssi.it

Connie U. Smith
Performance Engineering Services
L&S Computer
Technology, Inc. Austin, TX, USA
www.spe-ed.com

Catia Trubiani
Gran Sasso Science Institute
L'Aquila, Italy
catia.trubiani@gssi.it

ABSTRACT

Identifying performance problems in modern software systems is nontrivial, even more so when looking at specific application domains, such as cyber-physical systems. The heterogeneity of software and hardware components makes the process of performance evaluation more challenging, and traditional software performance engineering techniques may fail while dealing with interacting and heterogeneous components. The goal of this paper is to introduce a model-based approach to understand software performance problems in cyber-physical systems. In our previous work, we listed some common bad practices, namely software performance antipatterns, that may occur. Here we are interested in shedding light on these antipatterns by means of performance models, i.e., queuing network models, that provide evidence of how antipatterns may affect the overall system performance. Starting from the specification of three software performance antipatterns tailored for cyber-physical systems, we provide the queuing network models capturing the corresponding bad practices. The analysis of these models demonstrates their usefulness in recognizing performance problems early in the software development process. This way, performance engineers are supported in the task of detecting and fixing the performance criticalities.

CCS CONCEPTS

• **General and reference** → Performance; Metrics; • **Computer systems organization** → Embedded and cyber-physical systems.

KEYWORDS

Queuing Networks, Software Performance Antipatterns, Cyber-Physical Systems

ACM Reference Format:
Riccardo Pinciroli, Connie U. Smith, and Catia Trubiani. 2021. QN-based Modeling and Analysis of Software Performance Antipatterns for Cyber-Physical Systems. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427921.3450251

ICPE '21, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450251

1 INTRODUCTION

In the software development process, there is a high interest in the early validation of requirements, especially for performance-related characteristics that have been recently considered as the new system correctness [16]. The cost of fixing errors has been demonstrated to escalate exponentially as the project matures during the various phases of its life cycle [36].

Software Performance Engineering (SPE) [9, 28, 29] aims to produce performance models early in the development cycle. Solving such models produces predictions that can trigger the process of refactoring the system design to meet performance requirements[29]. In the last years, several strategies have been successfully adopted to automate the modeling and analysis of software performance [15], and optimization techniques [1]. However, the problem of interpreting performance analysis results is still critical, especially when considering application domains where the heterogeneity of software and hardware components may cause traditional SPE approaches to fail, e.g., cyber-physical systems (CPS) and internet of things (IoT).

In this paper, we focus on identifying performance issues in CPS, i.e., systems with heterogeneous software and hardware components. Our goal is to identify the system performance flaws by localizing the weakest points and rapidly fixing them. To achieve this objective, we make use of software performance antipatterns [33–35], recently customized for CPS [30]. The rationale behind this choice is that software performance antipatterns include the description of (i) problems leading to performance issues and (ii) best practices aimed to get performance improvements. Consider the *Blob* performance antipattern which occurs when a single component monopolizes the computation managing most of the work and becomes a system bottleneck. To solve this bad practice, it is necessary to improve the management of the system workload by delegating work to surrounding components and running the computation in a distributed fashion.

In the context of CPS, we focus on the following three software performance antipatterns: (i) *Are We There Yet?*, i.e., requests using computing resources to check the occurrence of some events; (ii) *Is Everything Ok?*, i.e., requests verifying the status of computing resources; (iii) *Where Was I?*, i.e., processes forgetting their current state and recalculating it. To model and study these antipatterns, we use the queuing network (QN) formalism [21] since it has been demonstrated to be effective in real-world scenarios [39]. The objective of our research is to show the usefulness of QN models in recognizing and analyzing performance problems that can be traced back to the occurrences of performance antipatterns. The main contributions of this paper is summarized as follows:

- the specification of QN models expressing the peculiarities of three software performance antipatterns for CPS;
- the injection of software performance antipatterns for CPS and their experimentation on a real case study;
- empirical evidence on the benefit of detecting and solving software performance antipatterns for CPS.

The rest of the paper is organized as follows. Section 2 provides some background on the three software performance antipatterns that we consider in this paper and briefly presents a motivating example, i.e., smart parking. Section 3 describes queuing networks that model the antipatterns applied to an abstract example (including a device and a server) and evaluates them experimentally to show the impact of antipatterns on the system performance. Section 4 assesses software performance antipatterns in the considered case study, i.e., a net of continuously-monitored sensors. Threats to validity are argued in Section 5. Section 6 briefly reviews related work. Concluding remarks and possible directions for future work are outlined in Section 7. All experiments and replication data are publicly available: https://doi.org/10.5281/zenodo.4495665.

2 PRELIMINARIES

In this section we briefly review the background concepts of software performance antipatterns and cyber-physical systems through a motivating scenario, i.e., a "smart" parking scenario.

2.1 Software Performance Antipatterns

Table 1 reports a brief description of software performance antipatterns defined in the context of Cyber-Physical Systems, further details are available in [30]. The first column shows antipatterns' names, followed by the textual explanation of the performance problems the corresponding antipattern triggers, and lastly the foreseen solutions for improving the system performance. Examples of each are in the following subsection.

Table 1: Software Performance Antipatterns in Cyber-Physical Systems [30].

Antipattern	Problem	Solution
Are we there yet?	The problem is the frequency and overhead of the checking relative to the time it takes for an event to occur.	Adjust the polling interval delay.
Is Everything OK?	It refers to repeatedly checking the CPS platform status, such as the remaining battery life, storage space, etc.	Change the platform status checking, e.g., based on pre-defined events, states, or time.
Where Was I?	It refers to processes that do not remember state information and there is excessive overhead to recalculate the state.	Save state, check if previous results apply, invoke context-dependent actions, or use a custom-designed solution.

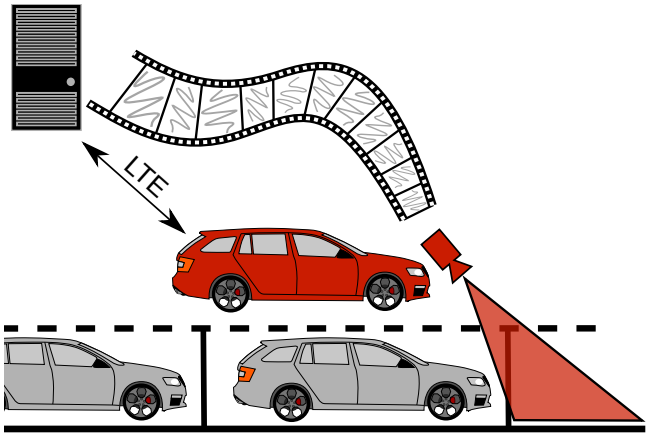


Figure 1: Sketch of the Smart Parking System [5].

2.2 Smart Parking System

Figure 1 provides a bird's eye view of a CPS that we use as our motivating example, namely the Smart Parking System, further details are available in [5]. The goals of this case study are: (i) evaluate the performance of the system design to provide information to cars looking for an empty parking spot; (ii) obtain predictive analytics from image data in a timely manner; (iii) determine the best design to meet the stated performance requirements.

From a performance engineering perspective, there are several performance antipatterns that could occur. For example, if the server polls the cars to see if they have new video available (i.e., *Are We There Yet?* performance antipattern) the polling interval may cause performance problems. If the time interval is too small, then the car is continuously interrupted, the server is busy with overhead rather than real work, and the overall system performance may suffer. If the time interval is too long, video may become stale before the server acts on it.

Another example is when the server frequently contacts all cars to confirm that their cameras are functioning correctly (i.e., *Is Everything OK?* performance antipattern). This implies that the retrieval of images is delayed and cars may have an unexpected delay in receiving parking results. As opposite, if it is in charge of cars to communicate any camera malfunction, then fewer messages are exchanged and this may be beneficial for the overall system function.

A final example is the possibility that the server does not remember previous parking results and re-starts the video analysis (i.e., *Where was I?*). If instead the server remembers "objects of interest" such as where parking spots were available, it could first make a quick check to see if it is still available. If the server does not remember previous results it wastes considerable time recalculating and the overall system performance suffers.

These scenarios motivate the value of automating the detection and solution of these bad practices. To this end, Section 3 presents QN models that show the performance effects of these performance antipatterns, and Section 4 demonstrates their usefulness in the context of a more complex case study, i.e., a network of sensors exchanging sensitive data.

3 OUR APPROACH

In this section, we describe our methodology to model performance antipatterns introduced in [30] using queuing networks [21]. For illustration purposes, here we intentionally adopt a simple and abstract software system composed of only a device and a server, and it represents our baseline (see Section 3.1). Modeling of software performance antipatterns is described decorating the baseline model in Section 3.2, and their impact on the system performance is analytically evaluated and discussed in Section 3.3.

3.1 Baseline

To investigate the effect of software performance antipatterns on a system, let us consider a single-class queuing network model with a delay station and two queueing stations (i.e., device and server), as showed in Figure 2. The model describes a batch (closed) system with a workload regulated by a think time, i.e., Z, where a *device* collects data (e.g., images, noise, temperature) that is processed by a *server*. Data collection and processing follow an exponential distribution with average service demand D_{dev} and D_{ser}, respectively. Both stations use a *Processor Sharing* queuing strategy (i.e., all requests receive an equal amount of the available service capacity) and the number of requests in the system, N_{req}, is fixed. Input model parameters are reported in Table 2, specifically we consider 10 requests (N_{req}) with 0 think time (Z), and service demands 0.02 (D_{dev}) and 0.04 (D_{ser}) for device and server, respectively. We use service demands since they account for both the service time and the number of visits to each station.

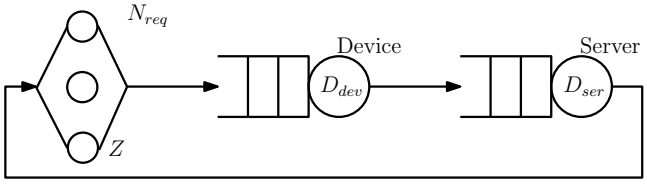


Figure 2: The queuing network model of the system used as baseline to study software performance antipatterns.

Table 2: Input parameters and performance indices of the queuing model in Figure 2 solved using MVA [21].

Parameters		Indices	
N_{req}	10	R_{sys}^{dflt}	0.4
D_{dev}	0.02	U_{dev}	0.5
D_{ser}	0.04	U_{ser}	1
Z	0	–	–

Table 2 also reports the baseline performance indices obtained using mean value analysis (MVA) [21]. These indices are considered in the sequel of the section to further investigate software performance antipatterns. More specifically, we plot in Figure 3 the performance indices of the considered system: (i) the default (dflt) system response time of requests – R_{sys}^{dflt}; and the usage of resources, i.e., (ii) the utilization of the device – U_{dev} and (iii) the utilization of the server – U_{ser}. These indices undergo a variation of the service demand of the *Server* station, D_{ser}, varying from 0 to 0.08, and the service demand of *Device*, D_{dev}, is instead set to 0.02. Note that the bottleneck of the system is the *device* when $D_{ser} < 0.02$, the *server* otherwise. The vertical dashed line shows the D_{ser} value used to determine the baseline performance (see Table 2).

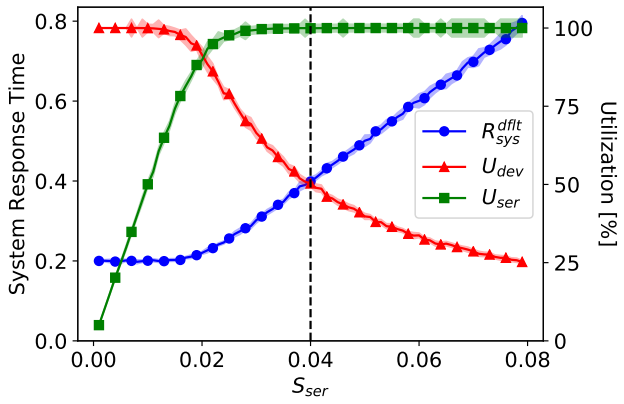


Figure 3: Model-based performance analysis results of the baseline queuing network.

3.2 Modeling

The three performance antipatterns introduced in [30] are modeled using the queuing network formalism. For illustration purposes, we adopt the baseline model in Figure 2 and we extend it to account for performance antipatterns. For the sake of clarity and without loss of generality, in this section, we assume that performance antipatterns affect the device station only. Note that all considerations and findings are easily reproducible if the server station is alternatively considered as target station to accommodate antipatterns.

3.2.1 Are We There Yet? Requests that use computing resources to check if an event has occurred are modeled by extending the QN in Figure 2 with a new request class (*checking*), as in Figure 4. Specifically, N_{chk} requests of the new *checking* class are initialized in the system (i.e., one for each event that must be monitored), each of which spends an exponentially distributed time with average Z_{chk} in the delay station. This way, we model the invocation of a checking request every time a certain (monitored) event happens. The *Device* service demand of the *checking* follows an exponential distribution with average D_{chk}. As stated in [30], the checking overhead is not negligible and requires many resources, i.e., $D_{chk} \simeq D_{dev}$. We recall that, for the sake of simplicity, we assume that the server is not affected by this software performance antipattern in the considered example. Therefore, the *checking* requests do not visit the *Server* station, they are routed back to the delay station.

3.2.2 Is Everything OK?. Similar to the *Are We There Yet?* software performance antipattern, the *Is Everything OK?* one is modeled by adding another request class (as in Figure 5). This time, the *checking* requests are invoked over and over to check the status of resource

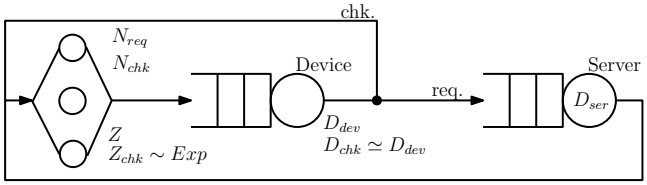


Figure 4: QN model of the *Are We There Yet?* software performance antipattern. A new class that models *checking* requests (chk) is introduced and an exponential checking interval is used. The service demand of the new class is similar to the default one.

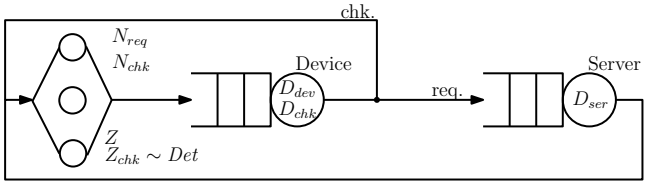


Figure 5: QN model of the *Is Everything OK?* software performance antipattern. A new class that models *checking* requests (chk) is introduced and a deterministic checking interval is used. The service demand of the new class is much smaller than the default one.

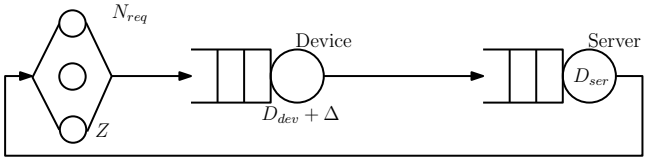


Figure 6: QN model of the *Where Was I?* software performance antipattern. The service demand of the affected resource is increased by Δ time units.

components (e.g., battery, storage). Assuming that this performance antipattern affects only the *device* of the considered system, a new checking request is sent to the device Z_{chk} time units after the previous check is completed. The periodic check is modeled with a *deterministic* think time since the interval between two consecutive polling is coded in the application. There are N_{chk} checking requests in the systems, i.e., the status of N_{chk} components of the device are checked periodically. The time to complete each status check, D_{chk}, is generally much smaller than the time required to process default requests (i.e., $D_{chk} \ll D_{dev}$).

3.2.3 Where Was I?. A process that loses its state must resume the execution from a past predefined state. This is modeled by increasing the service demand of the process at the station affected by the antipattern. Assuming that this performance antipattern affects the device of the system in Figure 2, the state loss is modeled by adding Δ time units to the device service demand, i.e., $D_{dev} + \Delta$, as shown in Figure 6. The value of Δ represents the *average* time spent by the device to recalculate the state during each visit to the *device*. Δ may be a minor recalculation; however, there may be cases (e.g., connectivity issues) that require extensive processing to recalculate the state [30].

Table 3: Input parameters of performance antipatterns when applied to the baseline system in Figure 2.

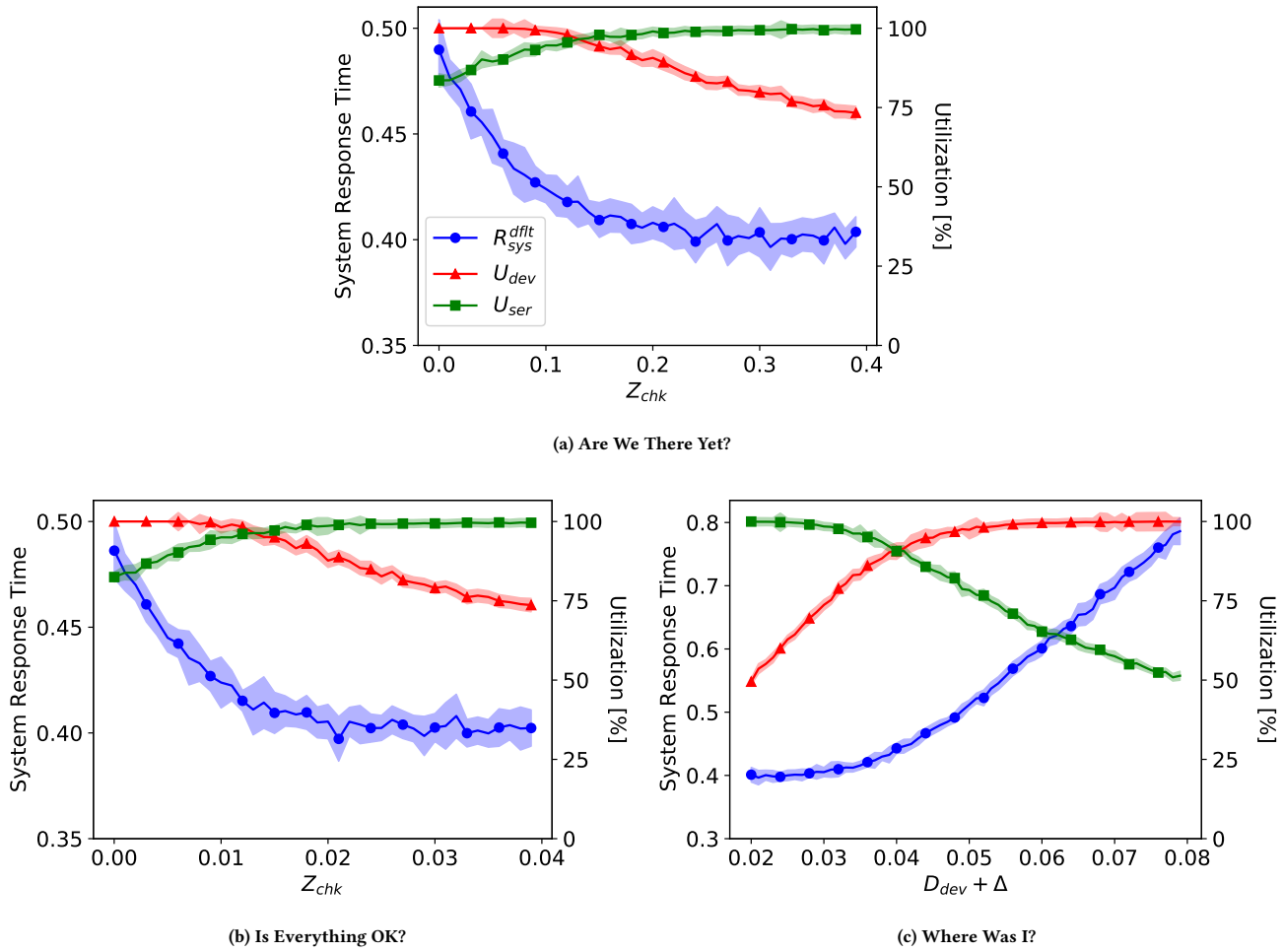
Antipattern	N_{chk}	D_{chk}	Z_{chk}	Δ
Are We There Yet?	10	0.01	(0 , 0.4)	–
Is Everything OK?	10	0.001	(0 , 0.04)	–
Where Was I?	–	–	–	[0 , 0.06)

3.3 Analysis

QN models presented and discussed in Section 3.2 are simulated, with Java Modelling Tools (JMT) [3], to investigate the effect of software antipatterns on the performance of the baseline system in Figure 2. Input parameters adopted for this investigation are reported in Table 3 and explained hereafter.

3.3.1 Are We There Yet? Parameters of the *checking* class introduced to model this performance antipattern (see Figure 4) are shown in the first row of Table 3. The number of checking requests is set to 10, i.e., 10 requests are sent to the device checking for some events. The service demand of checking requests is estimated to be half the time required to execute default requests (i.e., D_{dev}), i.e., polling events require many resources to be processed. The think time of checking requests varies in the interval $(0, 0.4)$ to evaluate how more frequent checking requests affect the performance of the system. Results are depicted in Figure 7(a), where the blue solid line is the system response time of default requests (plotted on the left y-axis), the red and partially dotted line is the device utilization, and the green dotted line is the server utilization (both plotted on the right y-axis). Shaded areas represent the 99% confidence interval. As expected, the system response time of default requests decreases when the checking interval increases. Longer checking intervals reduce the usage of the device and allow fast processing of default requests on the device. Baseline performance (i.e., $R_{sys}^{dflt} = 0.4$, as in Table 2) are observed when the checking interval is larger than 0.2 time units. Summing up, checking the occurrence of an event too often leads to performance overhead (perceived as an increase in the utilization of a resource) that prevents other resources from accomplishing their work, most likely switches the system bottleneck, and slows down the overall computation.

3.3.2 Is Everything OK?. Parameters that characterize the system in Figure 2 when it is affected by the *Is Everything OK?* software antipattern are provided in the second row of Table 3. The number of components whose status is repeatedly checked is set as follows, i.e., $N_{chk} = 10$. The status check requires fewer resources than the *Are We There Yet?* case since now the device must only return the status of the checked component. Therefore, D_{chk} is estimated to be one order of magnitude smaller than the previous case (i.e., $D_{chk} = 0.001$). The system performance are studied against the checking interval and shown in Figure 7(b). The performance behavior is similar to the one observed for the *Are We There Yet?* case (this is expected due to the similarity of the two antipatterns [30]). However, it is worth noting that baseline performance is observable with a short checking interval, i.e., larger than or equal to 0.02 time



(a) Are We There Yet?

(b) Is Everything OK?

(c) Where Was I?

Figure 7: Effect of the three performance antipatterns on the baseline system.

units. This is due to the short processing time required to retrieve the status of device components. Summing up, checking the status of a resource leads to a variation for the performance indices that is very similar to the *Are We There Yet?* antipattern (i.e., switching the system bottleneck and increasing/decreasing the utilization of resources). However, in this case, the performance overhead is generated by a high frequency of checking a resource, not from the checking activity itself that instead is rather small.

3.3.3 Where Was I?. The only parameter that is changed when modeling this performance antipattern is the service demand of the considered resource (i.e., the device, in this case). Specifically, a value Δ is added to the original service demand to model the processing time required to recalculate the state. As shown in Table 3, we consider the Δ value to vary from 0 to 0.06. This way, the system performance is evaluated against the device service demand in the range $[D_{dev}, D_{dev} + 0.06)$. Results are shown in Figure 7(c). When the *Where Was I?* antipattern does not affect the device (i.e., $\Delta = 0$) the server is the bottleneck of the system, i.e., $U_{ser} > U_{dev}$, and the system response time of the default request is the same observed in Table 2. The system response time of the default request increases with the device service demand ($D_{dev} + \Delta$). The device is the bottleneck of the system for $\Delta > D_{dev}$, i.e., restoring the state requires extensive processing.Summing up, when recalculating a state becomes more expensive than the actual computation, the response time can dramatically increase, and the system bottleneck switches.

4 CASE STUDY

In this section, we describe the CPS used to study the effect of software performance antipatterns in a realistic scenario.

4.1 Description

The resiliency of cyber-physical systems is essential and it is challenging to ensure that new systems meet resilience requirements without sacrificing performance. This case study shows how both security and performance can be analyzed before implementation,

Table 4: Input parameters for the sequence diagram in Figure 8.

Task	Processing time (msec)
Encrypt	0.96
Database Table Insert	0.6
Filter	0.21
Database Table Lookup	0.6
Decrypt	0.96
Predictive Analytics	0.32

and how the performance models can quantify the effect of performance antipatterns that may be present in the design.

The case study analyzes an existing data acquisition (sensorNet) system along with the machine control actions triggered by sensor values. We predict its performance when encryption is added to ensure that data is securely transferred between internal processors and the database. The original study [32] compared three options: one added security/encryption with the existing, basic sensors where the encryption and filtering happen on the (internal) controllers; the second evaluated replacing the basic sensors with smart sensors that do their own encryption; the third optimized the encryption algorithms to improve the basic sensor scenario. All used a cloud-hosted database for storing the data. This case study focuses on the original existing sensor version using a local database server and examines the performance effect with the presence of added performance antipatterns.

The encryption and decryption is based on an open source version of the advanced encryption standard (AES), a symmetric encryption algorithm developed by Daemen and Rijmen [10]. The encryption services are processed by the AES algorithm on the controller, filtered and then analyzed; the data is stored in the database. There is a table look up based on recent posts to the database which provides correlation to recent movement discovered by the sensor.

The processing steps for the *Analysis* scenario are shown in the sequence diagram reported in Figure 8. The antipatterns have been added to the case study as follows:

- *Are We There Yet?* is modeled with a workload (Polling) that executes at regular intervals to check if a sensor value has arrived.
- *Is Everything OK?* is modeled in a separate scenario (sequence diagram not shown) that executes at regular intervals to check the status of multiple platform resources, e.g., battery, buffer, sensors, etc.
- *Where Was I?* is modeled as the step refreshState just before the predict processing step.

The resource requirements are reported in Table 4. For example, we see that the *Encrypt* and *Decrypt* tasks require the most processing, in fact their processing time is 0.96 milliseconds, whereas other tasks show lower values.

4.2 Performance Models

In this section, we model the system described in Section 4.1 using execution graphs (EG) [29] (solved with SPE·ED) and queuing networks combined with Petri Nets (PN) [9] (solved with JMT). The

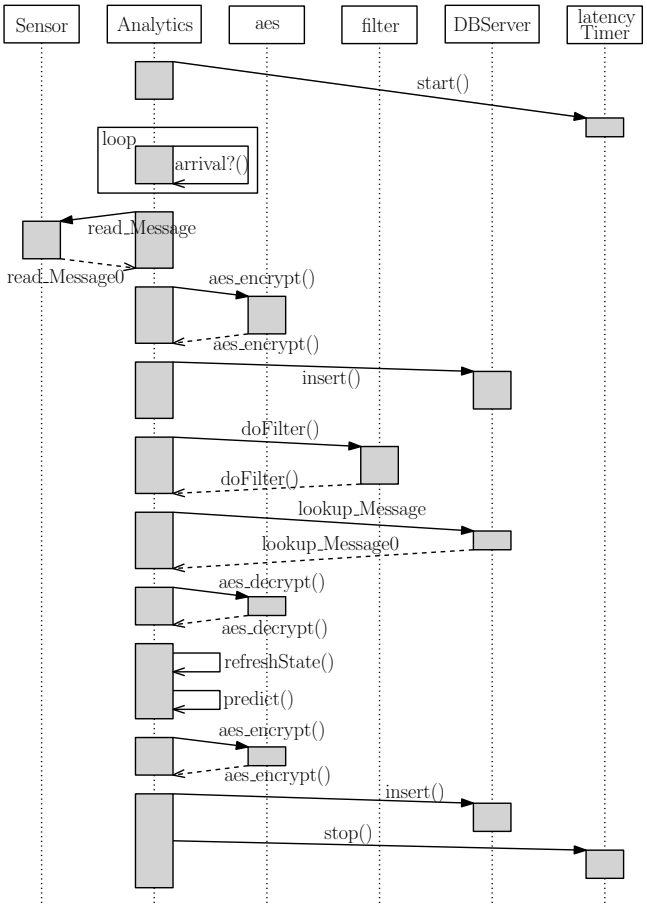


Figure 8: Sequence Diagram of the *Analysis* scenario.

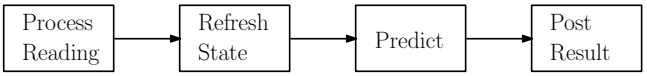


Figure 9: Steps in the Analysis sequence diagram.

validity of the QN+PN model is assessed by comparing its results with those obtained by solving the EG model. EG are a well-known formalism and they have been assessed in the SPE domain [29].

4.2.1 Execution Graphs. The system described in Section 4.1 is modeled by execution graphs of four workloads (i.e., Analysis, Actor, Status, and Polling). The workloads cycle through the Controller (central server) and the delay servers. After completion, workloads go to the think device where there is an exponential delay before the workload re-enters the system. The execution graph model is solved with SPE·ED [29], a tool designed specifically to support SPE methods and models, that provides as output performance analysis results.

Each workload of the system is modeled by a SPE·ED scenario derived from a sequence diagram (e.g., Figure 8 for the Analysis scenario). The interactions in the sequence diagram become steps in the corresponding SPE·ED scenario. The model for the steps

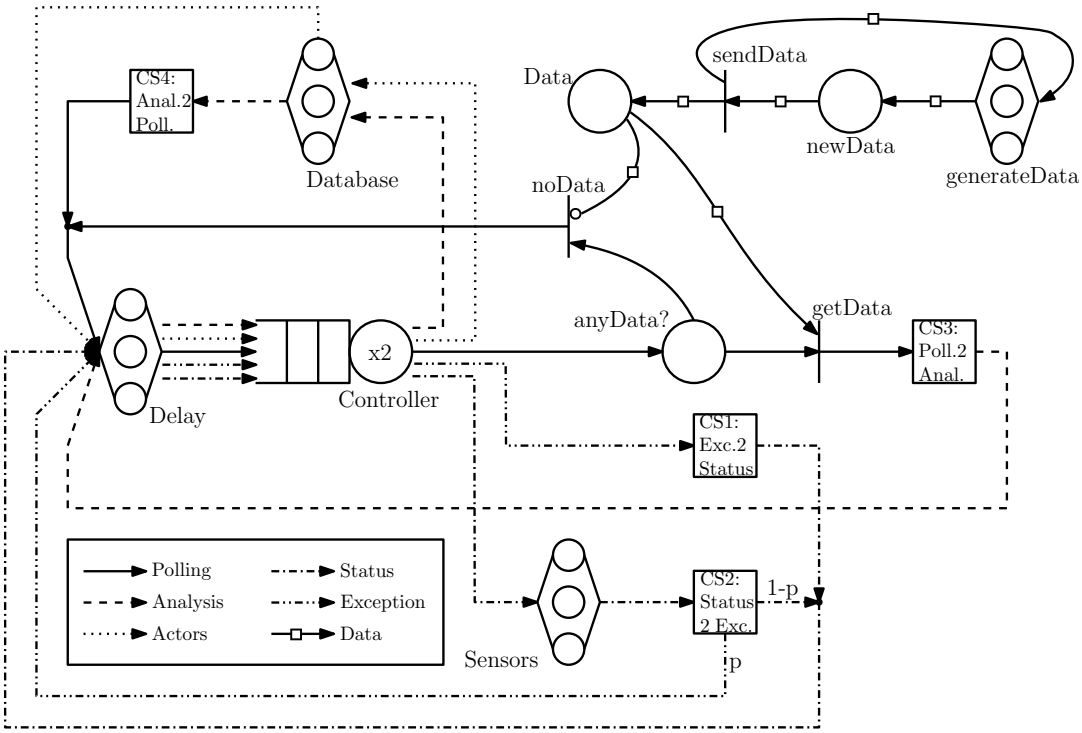


Figure 10: Multi-formalism model (QN and PN) of the CPS described in Section 4.1.

in the Analysis sequence diagram starting with the read-message interaction is straightforward as shown in Figure 9.

Where Was I? is an inserted processing step (Refresh state) just before predictive analytics. *Is Everything OK?* is the Status scenario with its own processing steps. *Are We There Yet?* is the Polling Scenario that repeatedly checks if a sensor reading has arrived. The Actor scenario was included to model the processing that occurs to correlate sensor readings and issue commands to the controlled device. It does not include antipatterns.

4.2.2 Queuing Networks. The CPS presented in Section 4.1 is studied using a multi-class and multi-formalism model. Besides queueing networks (QNs), we also make use of petri nets (PNs) to model the intricate synchronization and coordination details of this case study. Note that performance antipatterns are modeled using only the QN formalism as described in Section 3. Such a model is solved and analyzed using JSIMgraph, i.e., the simulator of JMT [3]. JSIMgraph discards the initial transient and automatically stops when the desired confidence interval is observed for all indices.

The multi-formalism model is shown in Figure 10. Five job classes are considered (i.e., *Polling*, *Analysis*, *Status*, *Exception*, and *Actors*) and they are represented by different line styles in the model. All these job types pass through a *Delay* station (i.e., the reference station) and the *Controller*, modeled as a queue station with two servers and a processor sharing (PS) queuing strategy. Service demands of all job classes follow an exponential distribution and the average demand for each station is shown in Table 5. For example, the average service demand associated to the Polling task at the Controller is 0.1 milliseconds. The only exception is the *Status* class that spends a deterministic time at the *Delay* station. This is due to the nature of this job class, i.e., a periodic check implemented in the system.

Petri Nets are used to model the generation of data from sensors. In fact, there is an extra job class, namely *Data* (i.e., dashed orange line), that represents the data generated by sensors and processed by the system. This way, we model the system polling the sensors for data to be analyzed. The number of sensors sending data to the system is N_{Data}, i.e., the number of Data jobs initialized in the *generateData* delay center. Sensors generate and send new data to the system every Z_{Data} time units. Then, data are collected in the Data PN place that can host only one request at a time. A sensor net system such as this *must* be able to process arriving data readings as quickly as they arrive. It is not acceptable to drop data. In this model we use the power of the multi-formalism model to track the number of failures: arrivals that would not be handled in a timely manner. In the model, if new data arrives before the previous one is polled by the system, then the oldest data is dropped, only the most recent one is kept, and the number of failures is reported by the model. In a sensor net system even one failure is unacceptable, but it is vital to know if failures could occur when antipatterns exist.

A Polling job (i.e., solid line) models requests generated by the controller to check if new sensor data is available. It goes to the *AnyData?* PN place after being processed by the controller. Here, it checks if data, generated by a sensor, is available in the *Data* PN place. If data is found, then the *getData* transition fires and the Polling job is switched to an Analysis job. This way, we model the impossibility for Polling and Analysis jobs to coexist in the system.

Table 5: The average service demand [msec] of all job classes at each station. All service demands follow an exponential distribution except the Status class at the Delay station, i.e., deterministic. Alongside the reference stations (i.e., *Delay* and *generateData*), the number of jobs for each class is shown in parentheses. The dash (i.e., –) means that the job class is not served by the station.

Station	Polling	Analysis	Status	Exception	Actors	Data
Controller	0.1	3.41	0.1	5	2.16	–
Database	–	1.8	–	–	1.2	–
Sensors	–	–	1	–	–	–
Delay	0.1 ($N = 1$)	0 ($N = 0$)	0.1 ($N = 1$)	0 ($N = 0$)	30 ($N = 5$)	–
generateData	–	–	–	–	–	60 ($N = 8$)

Table 6: Execution graph (EG) and queuing network (QN) results to assess the correctness of the adopted model.The 99% confidence interval of JMT simulations is shown in parenthesis. The utilization error is the distance between the observed usages. The system response time error is a mean absolute percentage error. Errors are computed wrt. average values.

	Utilization			System Response Time		
Job Class	EG [%]	QN [%]	Error (Diff.) [%]	EG [msec]	QN [msec]	Error (MAPE) [%]
Analysis	17.4	17.8 (± 0.41)	0.4	5.53	5.35 (± 0.10)	3.18
Status	3.9	4.1 (± 0.08)	0.2	1.17	1.11 (± 0.02)	5.05
Actors	16.1	15.8 (± 0.46)	0.3	3.51	3.64 (± 0.07)	3.85
Polling	10.0	10.9 (± 0.30)	0.9	2.06	2.18 (± 0.04)	5.72

If no data has been generated yet, then the *noData* transition fires and the Polling job goes back to the reference station where it stays for $Z^{Polling}$ time units before repeating the cycle.

The Analysis job (i.e., dashed line) models data generated by sensors that must be analyzed by the controller and stored in the database. It spends $Z_{Analysis} = 0$ time units in the *Delay* center. Hence, it is processed by the *Controller* and by the *Database* (represented as an infinite server) with service demands shown in Table 4. After the *Database* processing, the Analysis job is switched back to the Polling class and goes to the reference station.

A Status job (i.e., dash dotted line) represents requests aimed to check the status of sensors. It is processed by the controller after it spent Z_{Status} time units in the *Delay* station, i.e., a *Status* job is generated every Z_{Status} time units. It goes to the *Sensors* (i.e., infinite server) where it is served. With probability $1 - p$, no issue is detected and the *Status* job goes back to the *Delay*. An Exception (i.e., dash dot dotted line) is raised with probability p and it must be immediately (i.e., $Z_{Exception} = 0$ time units) handled by the *Controller*. An exception raise is a rare event that is modeled by a small value of p. Once the Exception job is processed, it is switched back to the Status class and it goes to the *Delay* to repeat the cycle.

An Actor (i.e., dotted line) is an entity that interacts with the controller and the database, and it is used to see the effect of performance antipatterns on the system. After waiting Z_{Actor} time units in the *Delay*, the Actor job goes first through the *Controller*, then to the *Database*, and finally goes back to the *Delay* station.

4.2.3 Model-based Performance Analysis. Results observed by solving the QN model in Figure 10 with JMT are compared to those obtained solving the EG model with SPE·ED. We recall that the input parameters used for evaluating the model in Figure 10 are provided in Table 5.

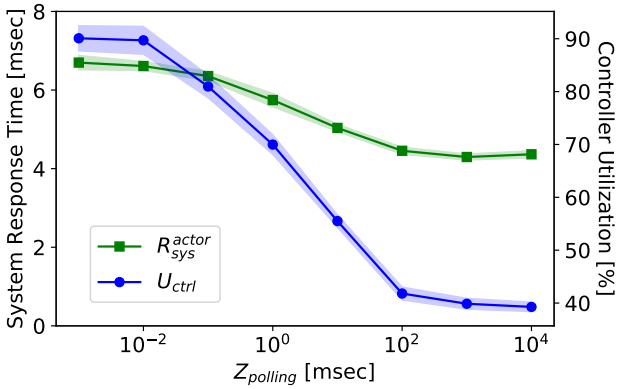
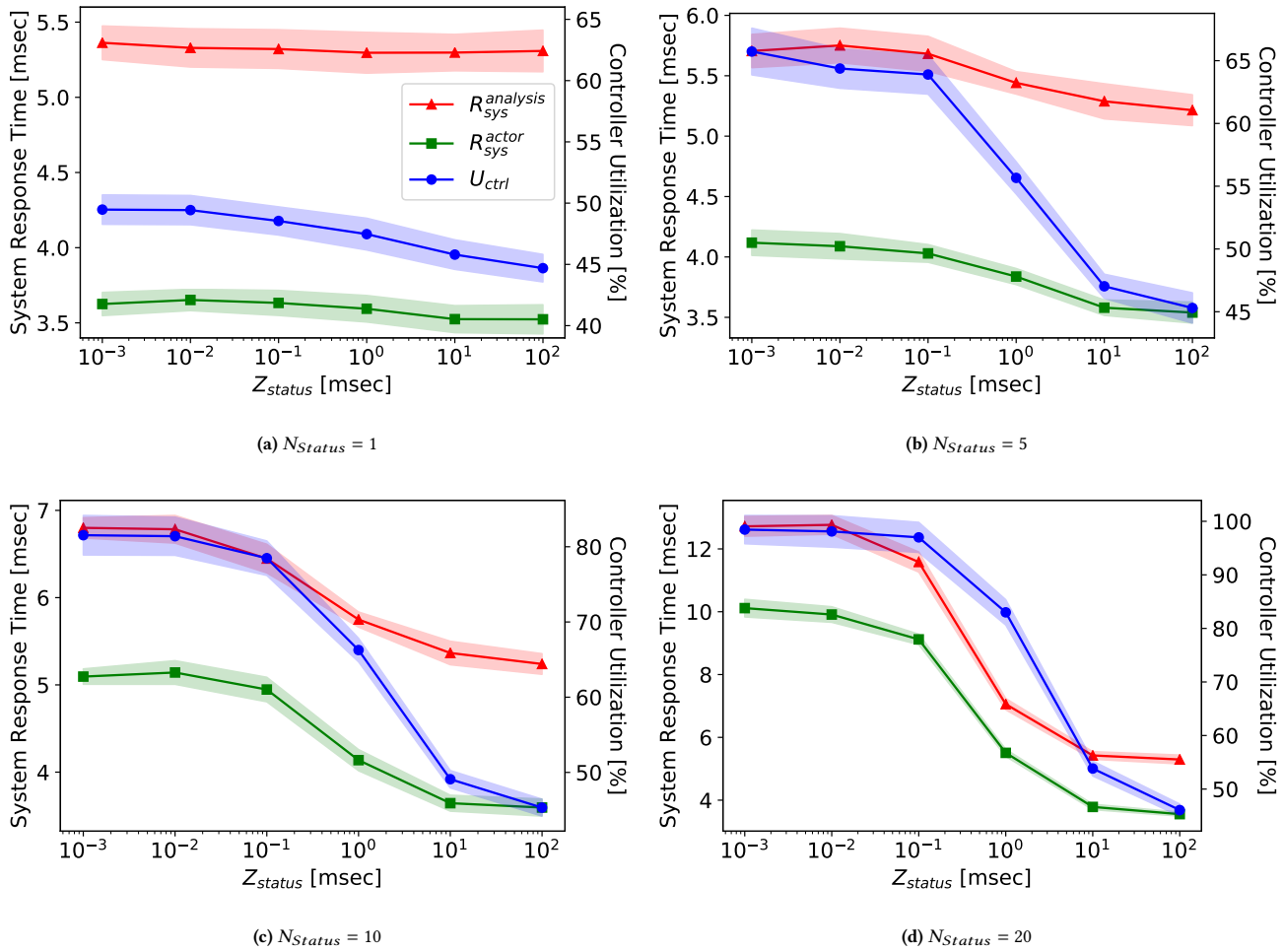


Figure 11: Effect of *Are We There Yet?* software performance antipattern on the sensor net system.

Table 6 shows the results. As performance indices of interest, we analyzed the utilization of the Controller and the system response time, both calculated for each job class. The *Exception* job class is not shown in Table 6 since we assume the probability that a status check generates an exception to be zero, i.e., $p = 0$. For both performance indices, the error made by using the multi-formalism model is derived considering the average performance value obtained solving the EG and QN models. The utilization error is the *difference* between the two utilization values, i.e., $|U_{\text{EG}} - U_{\text{QN}}|$. The error on the system response time is computed as the *mean absolute percentage error* (MAPE), i.e., $\frac{|R_{\text{EG}} - R_{\text{QN}}|}{R_{\text{EG}}} \cdot 100$.

Results demonstrate that the two models compare satisfactorily. For example, in the first row of Table 6 we see that for the Analysis



(a) $N_{Status} = 1$

(b) $N_{Status} = 5$

(c) $N_{Status} = 10$

(d) $N_{Status} = 20$

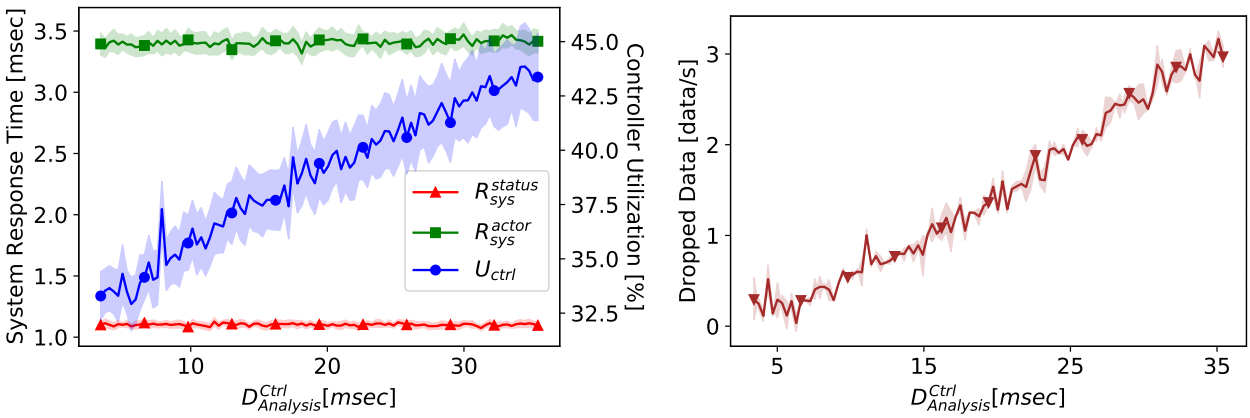
Figure 12: Effect of *Is Everything OK?* software performance antipattern on the sensor net system when checked devices (i.e., sensors) do not return exceptions. The performance of the *Controller* is evaluated for different numbers of *Status* jobs.

job class the utilization estimated with the EG model is 17.4%, the QN model instead evaluates 17.8% leading to an error of 0.4%. The Polling job class shows the highest error observed for utilization values, i.e., 0.9%. When considering the system response time, again for the Analysis job class, the EG models estimates 5.53 ms, whereas the QN model predicts 5.35 ms, hence we get an error of 3.18%. We do not expect the models to match exactly because of the different ways of modeling polling. Overall, albeit several approximations introduced in both models, the maximum error observed is smaller than 6% and it relates to the system response time only, whereas the errors for the utilization are much lower. The QN model adequately represents the case study performance and can be used to investigate CPS antipatterns.

4.3 Antipattern Experiments

Here, we analyze the effect of the three performance antipatterns on the sensor net system described in Section 4.1. To this end, we use the multi-formalism model shown in Figure 10 and inject the performance antipatterns as described in Section 3.

4.3.1 Are We There Yet? To investigate how the performance of the CPS degrades when the controller is affected by *Are We There Yet?* antipattern, we solve the multi-formalism model for different intervals between two consecutive polls. This is accomplished by changing the time that the Polling job spends in the delay center, i.e., $Z_{Polling}$. For the sake of simplicity, we assume that the controller has only one processor when studying this performance antipattern. Results are shown in Figure 11, where the 99% confidence interval (i.e., the shadowed area) for each measure is also depicted. The system response time for the Actors job class (left y-axis) and the controller utilization (right y-axis) are plotted against the Polling think time (log scale). Less frequent polling (i.e., large values of $Z_{Polling}$) allows decreasing the controller usage as well as the system response time of the Actors job class. When the polling interval is short (i.e., $Z_{Polling}$ is small) the controller looks for new



(a) System response time and utilization

(b) Dropped data

Figure 13: Effect of *Where Was I?* software performance antipattern on the sensor net system. Results are obtained considering only 1 Actor and 1 Sensor in the system. Other input parameters are the same as those in Table 5.

data too often. This increases the controller usage tremendously (up to 2.5x) and other jobs (i.e., Actors) experience a slower (i.e., 1.3x) latency.

4.3.2 Is Everything OK?. The effect of this performance antipattern on the CPS is depicted in Figure 12 varying the number of monitored sensors. Periodically, the controller contacts sensors to check that no problem has occurred. The negative effect of this performance antipattern is especially visible when the number of monitored sensors is large and the period between two consecutive status checks is short. Note that, we assume the probability that a status check raises an exception, p, to be zero. This way, the controller does not handle exceptions that require high processing capacity (see Table 5). The system response time of Analysis and Actors job classes (left y-axis), the controller utilization (right y-axis), and their 99% confidence interval (shadowed areas) are plotted against the period between two status checks (log scale), i.e., the time spent by a Status job at the delay center. Each simulation is run for a different number of monitored sensors (i.e., 1, 5, 10, and 20). If a single sensor is monitored by the controller, see Figure 12(a), the performance degradation due to the antipattern is negligible, independently of how frequently the controller contacts the sensor asking for its status. When the controller monitors the status of many sensors, a short Z_{Status} makes the controller utilization higher (i.e., up to 2x, the controller saturates for $N_{Status} = 20$) and the system response time of other jobs served by the controller (i.e., Analysis and Actor) dramatically deteriorates (i.e., 2.5x slower).

4.3.3 Where Was I?. The effect on system performance of a process the loses its state and must retrieve it is depicted in Figure 13. More specifically, in Figure 13(a), the system response time of Status and Actors job classes (left y-axis), the utilization of the controller (right y-axis), and their 99% confidence interval (shadowed area) are plotted against different values of $D^{Ctrl}_{Analysis}$, i.e, the service demand of Analysis jobs at the controller. The minimum value of $D^{Ctrl}_{Analysis}$ is the service demand shown in Table 5 (i.e., 3.42 ms). Longer values are considered to represent the delay (i.e., Δ in Figure 6) required to recalculate the (lost) state and the processing time required is large. Although a long $D^{Ctrl}_{Analysis}$ makes the controller utilization increase (i.e., 1.3x), it does not affect the latency of other jobs.

Figure 13(b) depicts the amount of data that would be dropped every second against the Analysis service demand at the controller. Since the system cannot poll and analyze data at the same time, an Analysis job that requires extensive processing to restore its state reduces the polling frequency and increases the amount of data that cannot be processed in a timely manner. Dropping data is unacceptable in a data acquisition system. If it happened in an aircraft traffic data system that cannot keep pace with traffic data in a congested area due to a *Where Was I?* performance antipattern consequences could be tragic.

5 THREATS TO VALIDITY

In this section, we discuss the main threats to validity exhibited by our approach.

First, we are aware that generalization of results (i.e., *external validity*) is not guaranteed, since our models have been applied to one case study only, however the sensor net has been already used as a representative example of CPS in software performance engineering research, as confirmed by [32]. We also studied abstract models of antipattern performance that apply to many other types of systems in section 3. We expect that it is possible to generalize our results and findings to situations that embody the bad practices described in [30].

Second, to mitigate threats to *internal validity*, we designed our experiments with the goal of having a direct manipulation on the performance indices of interest. For instance, the baseline queuing model (see Figure 3) shows input parameters that lead to 0.4 as utilization of the device. This choice is motivated by the illustration

purpose of studying variations when performance antipatterns are put in place (see Figure 7). Moreover, the choice of using Queuing Networks as the target notation for modelling antipatterns does not reduce the applicability of our approach. In the case study, we also used Petri Nets to model application peculiarities (i.e., process synchronization), and the resulting multi-formalism model is still valid to provide the evidence of antipatterns on the system performance. In general, any formalism can be adopted to model antipatterns as long as it is suitable to express execution times subject to variations. We plan to further experiment this point by investigating other languages to model antipatterns.

Third, to smooth *construct validity* threats, i.e., the assessment of the validity of the metrics used during our experimentation, we set that all simulations undergo a 99% confidence interval, so the accuracy of numerical results has been monitored.

6 RELATED WORK

The work presented in this paper relates to two main streams of research that we review in the following.

Software Performance Antipatterns. They have been defined in the literature as bad practices leading to performance issues [27, 29], and very recently customized for the CPS domain in [30]. Our recent work focused on investigating the performance antipatterns across the operational profile space [6], previously defined with a first-order logic representation, and later applied to multiple modelling notations, specifically in a general-purpose language (UML), a domain-specific language (Palladio), and an architecture description language (Æmilia). A first attempt of adopting software performance antipatterns in running systems is provided in [40], where the root causes of performance problems are isolated and matched with the specification of antipatterns. More recently, load testing and profiling data is exploited to detect software performance antipatterns when running java applications in [38]. In the broader context of matching the connections between (anti)patterns and quality attributes (such as reliability, security, etc.), several approaches e.g., [11, 14, 17, 25] are representative.

Performance Evaluation of CPS. In literature several approaches have been defined for the modelling of CPS (e.g., [18, 20, 26]) and its security-related aspects (e.g., [23, 37, 42]). Performance analysis, instead, was mainly focusing on real-time embedded systems, we refer the reader to [43] for a broader investigation. As opposite, CPS demands for a plethora of performance evaluation techniques [4], and there exists two macro classes: (i) the *analytical* and (ii) the *simulation* analysis. Analytical approaches use mathematical formulas or equations that are formal and rigorous, but they may fail to capture some system dynamics (e.g., unexpected events, uncertainties, transient states) that can be expressed in simulation environments (i.e., emulating the system behavior) at the cost of less scalability [13, 45, 46]. Co-simulation has also been more recently proposed with the goal of possibly integrating multiple and heterogeneous models [22]. In [24] a linear stochastic model is adopted to quantify the performance degradation of CPS when exposed to integrity attacks. In [12] analytical models are adopted to derive asymptotic and worst case scenarios for performance analysis. In [44] the performance evaluation is conducted through a control law that undergoes a trade-off analysis including privacy costs. In [7] Markov models are applied in the intelligent transportation system domain, and traffic is guided by model predictions. When focusing on the software performance engineering community, it is worth mentioning several frameworks that have been introduced in the literature for performance modeling and analysis of software systems. For example: (i) the Core Scenario Model (CSM) [41], (ii) Klaper [8], (iii) the Performance Model Interchange Format (PMIF) and Software-PMIF (S-PMIF) [31], (iv) the Palladio Component Model (PCM) [2], and (v) Descartes [19]. However, to the best of our knowledge, none of these frameworks is specifically tailored to modelling the detection and the fixing of software performance antipatterns in CPS [30], as in our investigation in this paper.

7 CONCLUSION AND FUTURE WORK

In this paper, we present a novel approach to model and analyze software performance antipatterns in the context of cyber-physical systems. Queuing networks are adopted as the performance modeling formalism of choice, and performance results confirm the usefulness of our models. For all the three software antipatterns considered in this paper, we propose a QN model to investigate its effect on the system performance. We always observe the system response time increasing and the system bottleneck switching. Modeling software antipatterns with QN allows keeping track of performance problems and quantitatively evaluating their effect on the system performance. A sensor net case study is proposed to quantify the performance impact of antipatterns, and experimental results demonstrate the effectiveness of our approach. When evaluating the system performance indices, we found that antipatterns may worsen the response time and resource utilization up to 2.5x. This data encourages detecting bad practices early in the design process.

As future work, we plan to automate the detection of software antipatterns by building a framework that (i) keeps track of the required information (e.g., frequency of checking the status of a resource), and (ii) exploits the performance analysis results as the basis for antipattern solutions. For instance, knowing the point where the system bottleneck switches is of key relevance to prevent it. Moreover, we are also interested in investigating the guiltiness of antipatterns wrt. requirements, i.e., how much antipatterns contribute to the violation of requirements. This way, we can prioritize those antipatterns to solve first. Finally, we plan to investigate other CPS case studies, possibly from an industrial context, in order to better investigate the performance impact of antipatterns in different application domains.

ACKNOWLEDGMENTS

This work has been partially funded by MIUR PRIN project 2017TWRCNB SEDUCE (Designing Spatially Distributed Cyber-Physical Systems under Uncertainty).

REFERENCES

[1] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. 2013. Software Architecture Optimization Methods: A Systematic Literature Review. *IEEE Trans. Software Eng.* 39, 5 (2013), 658–683.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 2009. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software* 82, 1 (2009), 3–22.

[3] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. 2009. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.* 36, 4 (2009), 10–15.

[4] André B Bondi. 2015. *Foundations of software and system performance engineering: process, performance modeling, requirements, testing, scalability, and practice.* Pearson Education.

[5] Tomás Bures, Vladimír Matena, Raffaela Mirandola, Lorenzo Pagliari, and Catia Trubiani. 2018. Performance Modelling of Smart Cyber-Physical Systems. In *Proceedings of the International Conference on Performance Engineering (ICPE).* 37–40.

[6] Radu Calinescu, Vittorio Cortellessa, Ioannis Stefanakos, and Catia Trubiani. 2020. Analysis and Refactoring of Software Systems Using Performance Antipattern Profiles. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE).* 357–377.

[7] Chen Chen, Xiaomin Liu, Tie Qiu, and Arun Kumar Sangaiah. 2020. A short-term traffic prediction model in the vehicular cyber–physical systems. *Future Generation Computer Systems* 105 (2020), 894–903.

[8] Andrea Ciancone, Mauro Luigi Drago, Antonio Filieri, Vincenzo Grassi, Heiko Koziolek, and Raffaela Mirandola. 2014. The KlaperSuite framework for model-driven reliability analysis of component-based systems. *Software and System Modeling* 13, 4 (2014), 1269–1290.

[9] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. 2011. *Model-Based Software Performance Analysis.* Springer.

[10] Joan Daemen and Vincent Rijmen. 2002. *The Design of Rijndael.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[11] Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, Alexander Chatzigeorgiou, and Elisa Yumi Nakagawa. 2019. What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes? *Inf. Softw. Technol.* 105 (2019), 1–16.

[12] Allan Edgard Silva Freitas and Romildo Martins da Silva Bezerra. 2015. Performance Evaluation of Cyber-Physical Systems. *ICIC Express Letters* 10, 2 (2015).

[13] Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nystrom, Levon Saldamli, David Broman, and Anders Sandholm. 2006. OpenModelica-A free open-source environment for system modeling, simulation, and teaching. In *Proceedings of the International Conference on Control Applications (ICoCTA).* 1588–1595.

[14] Matthias Galster and Paris Avgeriou. 2012. Qualitative Analysis of the Impact of SOA Patterns on Quality Attributes. In *Proceedings of International Conference on Quality Software (QSIC).* 167–170.

[15] Abel Gómez, Connie U Smith, Amy Spellmann, and Jordi Cabot. 2018. Enabling performance modeling for the masses: Initial experiences. In *Proceedings of the International Conference on System Analysis and Modeling (SAM).* 105–126.

[16] Mark Harman and Peter W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM).* 1–23.

[17] Geoffrey Hecht, Benjamin Jose-Scheidt, Clement De Figueiredo, Naouel Moha, and Foutse Khomh. 2014. An Empirical Study of the Impact of Cloud Patterns on Quality of Service (QoS). In *Proceedings of International Conference on Cloud Computing Technology and Science (CloudCom).* 278–283.

[18] Christian Heinzemann, Steffen Becker, and Andreas Volk. 2019. Transactional execution of hierarchical reconfigurations in cyber-physical systems. *Softw. Syst. Model.* 18, 1 (2019), 157–189.

[19] Nikolaus Huber, Fabian Brosig, Simon Spinner, Samuel Kounev, and Manuel Bähr. 2017. Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. *IEEE Trans. Software Eng.* 43, 5 (2017), 432–452.

[20] Kim Guldstrand Larsen. 2017. Validation, Synthesis and Optimization for Cyber-Physical Systems. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Vol. 10205. 3–20.

[21] E. D. Lazowska, J. Zahorjan, G. Scott Graham, and K. C. Sevcik. 1984. *Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Inc., Englewood Cliffs.

[22] Giovanni Liboni, Julien Deantoni, Antonio Portaluri, Davide Quaglia, and Robert De Simone. 2018. Beyond time-triggered co-simulation of cyber-physical systems for performance and accuracy improvements. In *Proceedings of the International Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO).* 1–8.

[23] An-Yang Lu and Guang-Hong Yang. 2020. Stability Analysis for Cyber-Physical Systems Under Denial-of-Service Attacks. *IEEE Trans. on Cybernetics* (2020), 1–10.

[24] Yilin Mo and Bruno Sinopoli. 2016. On the Performance Degradation of Cyber-Physical Systems Under Stealthy Integrity Attacks. *IEEE Trans. Automat. Contr.* 61, 9 (2016), 2618–2624.

[25] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.* 36, 1 (2009), 20–36.

[26] Pierluigi Nuzzo, Jiwei Li, Alberto L. Sangiovanni-Vincentelli, Yugeng Xi, and Dewei Li. 2019. Stochastic Assume-Guarantee Contracts for Cyber-Physical System Design. *ACM Trans. Embed. Comput. Syst.* 18, 1 (2019), 2:1–2:26.

[27] Trevor Parsons and John Murphy. 2008. Detecting Performance Antipatterns in Component Based Enterprise Systems. *J. Object Technol.* 7, 3 (2008), 55–91.

[28] Dorina C. Petriu, Mohammad Alhaj, and Rasha Tawhid. 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 (Lecture Notes in Computer Science, Vol. 7320).* Springer, 219–262.

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

[30] Connie U. Smith. 2020. Software Performance Antipatterns in Cyber-Physical Systems. In *Proceedings of the International Conference on Performance Engineering (ICPE).* 173–180.

[31] Connie U Smith, Catalina M Lladó, and Ramon Puigjaner. 2010. Performance Model Interchange Format (PMIF 2): A comprehensive approach to queueing network model interoperability. *Performance Evaluation* 67, 7 (2010), 548–568.

[32] Connie U. Smith and Amy Spellmann. 2017. Automated Performance Modeling for IoT Systems. In *L&S Computer Technology, Inc.* https://bit.ly/3jNeocC

[33] Connie U Smith and Lloyd G Williams. 2000. Software performance antipatterns. In *Proceedings of the International Workshop on Software and Performance (WOSP).* 127–136.

[34] Connie U Smith and Lloyd G Williams. 2002. New software performance antipatterns: More ways to shoot yourself in the foot. In *Proceedings of the International Conference on Computer Measurement Group (CMG).* 667–674.

[35] Connie U Smith and Lloyd G Williams. 2003. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Proceedings of the International Conference on Computer Measurement Group (CMG).* 717–725.

[36] Jonette M Stecklein, Jim Dabney, Brandon Dick, Bill Haskins, Randy Lovell, and Gregory Moroney. 2004. Error cost escalation through the project life cycle. *NASA Technical Report* (2004).

[37] Ashraf Tantawy, Sherif Abdelwahed, Abdelkarim Erradi, and Khaled Shaban. 2020. Model-based risk assessment for cyber physical systems security. *Computers & Security* 96 (2020), 101864.

[38] Catia Trubiani, Alexander Bran, André van Hoorn, Alberto Avritzer, and Holger Knoche. 2018. Exploiting load testing and profiling for Performance Antipattern Detection. *Inf. Softw. Technol.* 95 (2018), 329–345.

[39] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. 2005. An analytical model for multi-tier internet services and its applications. *ACM SIGMETRICS Performance Evaluation Review* 33, 1 (2005), 291–302.

[40] Alexander Wert, Jens Happe, and Lucia Happe. 2013. Supporting swift reaction: automatically uncovering performance problems by systematic experiments. In *Proceedings of the International Conference on Software Engineering (ICSE).* 552–561.

[41] C. Murray Woodside, Dorina C. Petriu, Dorin Bogdan Petriu, Hui Shen, Toqeer Israr, and José Merseguer. 2005. Performance by unified model analysis (PUMA). In *Proceedings of the International Workshop on Software and Performance, (WOSP).* 1–12.

[42] Zhiyan Xu, Debiao He, Huaqun Wang, Pandi Vijayakumar, and Kim-Kwang Raymond Choo. 2020. A novel proxy-oriented public auditing scheme for cloud-based medical cyber physical systems. *Journal of Information Security and Applications* 51 (2020), 102453.

[43] Ti-Yen Yen and Wayne Wolf. 1998. Performance estimation for real-time distributed embedded systems. *IEEE Trans. on Parallel and Distributed Systems* 9, 11 (1998), 1125–1136.

[44] Heng Zhang, Yuanchao Shu, Peng Cheng, and Jiming Chen. 2016. Privacy and performance trade-off in cyber-physical systems. *IEEE Network* 30, 2 (2016).

[45] Zhenkai Zhang, Emeka Eyisi, Xenofon Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits. 2014. A co-simulation framework for design of time-triggered automotive cyber physical systems. *Simulation Modelling Practice and Theory* 43 (2014), 16–33.

[46] Zhenkai Zhang, Joseph Porter, Emeka Eyisi, Gabor Karsai, Xenofon Koutsoukos, and Janos Sztipanovits. 2013. Co-simulation framework for design of time-triggered cyber physical systems. In *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS).* 119–128.