VERIFIED CONTROL AND ESTIMATION FOR CLOUD COMPUTING

by

ALEXANDROS EVANGELIDIS

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
January 2020

Abstract

In this thesis we propose formal verification as a way to produce rigorous performance guarantees for resource control and estimation mechanisms in cloud computing. In particular, with respect to control, we focus on an automated resource provisioning mechanism, commonly referred to as auto-scaling, which allows resources to be acquired and released on demand. However, the shared environment, along with the exponentially large space of available parameters, makes the configuration of auto-scaling policies a challenging task. To address this problem, we propose a novel approach based on performance modelling and formal verification to produce performance guarantees on particular rule-based auto-scaling policies. We demonstrate the usefulness and efficiency of our techniques through a detailed validation process on two public cloud providers, Amazon EC2 and Microsoft Azure, targeting two cloud computing models, Infrastructure as a Service (IaaS) and Platform as a Service (PaaS), respectively.

We then develop novel solutions for the problem of verifying state estimation algorithms, such as the Kalman filter, in the context of cloud computing. To achieve this, we first tackle the broader problem of developing a methodology for verifying properties related to numerical and modelling errors in Kalman filters. This targets more general applications such as automotive and aerospace engineering, where the Kalman filter has been extensively applied. This allows us to develop a general framework for modelling and verifying different filter implementations operating on linear discrete-time stochastic systems, and ultimately tackle the more specific case of cloud computing.

Acknowledgements

I am grateful to my supervisor Professor David Parker for the generous support and guidance throughout my research. His insightful comments and constructive feedback have been invaluable. My ideas and how they have been shaped have benefited from his wide knowledge and the opportunity to be in ongoing conversation with him.

I would also like to thank Dr Rami Bahsoon, who has also been a supervisor over the course of my studies, for providing me with valuable advice and guidance, and for encouraging me to embark on my PhD studies.

For their helpful comments and feedback on my research I also wish to thank Professor Ela Claridge and Dr David Oswald, my Thesis Group members.

Finally, I gratefully acknowledge the funding sources that allowed me to undertake my doctoral research, the Engineering and Physical Sciences Research Council (EPSRC) and the School of Computer Science, University of Birmingham.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Cloud computing has become the most prominent way of delivering software solutions, and more and more software vendors are deploying their applications in the public cloud. In cloud computing, one of the key differentiating factors between successful and unsuccessful application providers is the ability to provide performance guarantees to customers, which allows violations in performance metrics such as CPU utilisation to be avoided [2]. In order to achieve this, cloud application providers use one of the key features of cloud computing: *auto-scaling*, a resource provisioning mechanism, which allows resources to be acquired and released on demand.

While auto-scaling is an extremely valuable feature for application providers, specifying an *auto-scaling policy* that can guarantee no performance violations will occur is an extremely hard task, and *"doomed to fail"* [3] unless considerable care is taken. Furthermore, in order for a *rule-based* auto-scaling policy to be properly configured, there has to be an in-depth level of knowledge and a high degree of expertise, which is not necessarily true in practice [4, 2]. The rule-based method is the most popular, and is considered to be the *state-of-the-art* in auto-scaling an application in the cloud [5], and these challenges exist even in the case when a single auto-scaling rule needs to be specified. In a rule-based approach, the application provider has to specify an upper and/or lower bound on a performance metric (e.g. CPU utilisation) along with the desired change in capacity for this situation. For example, a rule-based method that will trigger an auto-scaling decision when CPU utilisation exceeds 60% might take the form: *if cpu_utilisation* > 60% *then add* 1 *instance* [6].

Lately, public cloud providers such as Amazon EC2 and Microsoft Azure have increased the flexibility offered to users when defining auto-scaling policies, by allowing combinations of auto-scaling rules for a wide range of metrics. However, this freedom of being able to specify multiple auto-scaling rules comes at the cost of an extremely large configuration space. In fact, it is exponential in the number of performance metrics and predicates, making it virtually impossible to find the optimal values for the auto-scaling variables [7].

In addition, an auto-scaling policy consists not only of performance metrics thresholds, but also of *temporal parameters*, which often seem to be neglected, despite their significance in configuring a good auto-scaling policy. These parameters include the time interval that the auto-scaling mechanism looks back to determine whether to take an auto-scale action, and the duration for which it is prohibited from triggering auto-scale actions after a successful auto-scale request (cool-down period). Since both of these parameters have to be specified by a human operator, it becomes a challenging task to understand the impact of these parameters on performance metrics of the application running on the cloud.

As noted in [8], auto-scaling policies *"tend to lack correctness guarantees"*. The ability to specify auto-scaling policies that can provide performance guarantees and reduce violations of Service Level Agreements (SLAs) is essential for more dependable and accountable cloud operations. However, this is a complex task due to: (i) the large configuration space of the conditions and parameters that need to be defined; (ii) the unpredictability of the cloud as an operating environment, due to its shared, elastic and on demand nature; and (iii) the heterogeneity in cloud resource provision, which makes it difficult to define reliable and universal auto-scaling policies. For example, looking at public cloud providers, one can observe that there is no guarantee on the time it will take for an auto-scale request to be served, nor whether the auto-scale request will receive a successful response or not.

Furthermore, in recent years, there have been several proposals to make the resource provisioning mechanisms which exist in the cloud more *autonomous* by integrating them with *Bayesian state estimation* algorithms, in order to minimise the role of the human operators [9, 10, 11]. In general, estimating the state of a continuously changing system based on uncertain information about its dynamics is a crucial task in many application domains ranging from control systems to econometrics. One of the most popular algorithms for tackling this problem is the *Kalman filter* [12], which essentially computes an optimal state estimate of a noisy linear discrete-time system, under certain assumptions, with the optimality criterion being defined as the minimisation of the mean squared error.

These proposals often come from advocates of the autonomic computing paradigm in which the Kalman filter is combined with a control system (i.e. controller), in order to provide an effective way of automating the resource-allocation decisions. The integration of a Kalman filter with a controller stems from the fact that the Kalman filter can be used as a *predictor* for predicting noisy performance parameters, such as the CPU utilisation. These predicted values are then passed as inputs to the controller, thus allowing for a proactive resource provisioning approach to be taken. Furthermore, if predictions about the future state of the system are not of interest, the Kalman filter can be used purely as an *estimator* for tracking the performance parameters of the "current" state of the system. This is a particularly advantageous/appropriate use case for the Kalman filter since it is very effective in filtering out the noise from the true signal (e.g. CPU utilisation data).

However, despite the fact that there are many advantages in making the resource provisioning mechanisms more sophisticated by integrating them with Kalman filters, there are significant challenges, in terms of their *verifiability*, which are associated with this decision. This is because, despite the robust mathematical foundations underpinning the Kalman filter, developing an operational filter in practice is considered a very hard task since it requires a significant amount of engineering expertise [13]. In particular, the underlying theory makes assumptions which are not necessarily met in practice, such as there being precise knowledge of the system and the noise models, and that infinite precision arithmetic is used [14, 15]. For example, avoidance of numerical problems, such as round-off errors, remains a prominent issue in Kalman filter implementations [14, 15, 16, 17].

The first contribution of this thesis addresses the challenges that exist in producing rigorous performance guarantees for rule-based auto-scaling policies by presenting novel approaches based on *quantitative verification*, which is a formal approach to generating guarantees about quantitative aspects of systems exhibiting probabilistic behaviour. In particular, we use *probabilistic model checking* and the PRISM tool [18, 19], where guarantees are expressed in quantitative extensions of temporal logic and numerical solution of probabilistic models is used to precisely quantify performance measures (e.g. probability of a performance metric exceeding a threshold). This approach provides a formal way of quantifying the uncertainty that exists in today's cloud-based systems and a means of providing performance guarantees on auto-scaling policies for application designers and developers. Another important novel aspect of our approach is the combination of probabilistic model checking with *Receiver Operating Characteristic (ROC)* analysis during empirical validation. This allows us not only to refine our original probabilistic estimates after collating real data and to validate the accuracy of our model, but also to obtain global Quality of Service (QoS) violation thresholds for the policies. We demonstrate the correctness and usefulness of this approach through an extensive validation, considering an Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) scenario running on the Amazon EC2 and Microsoft Azure cloud, respectively. We have made the models and data used to validate our models publicly available [20].

We then build on this work to develop novel solutions for the problem of verifying state estimation algorithms in the context of cloud computing. To achieve that, we first perform a detailed study on other fields, such as automotive and aerospace engineering, where the Kalman filter has been extensively applied. This allows us to develop a general framework for modelling and verifying different filter implementations operating on linear discrete-time stochastic systems. It consists of a modelling abstraction which maps the execution of a Kalman filter estimating the state of linear stochastic dynamical system to a discrete-time Markov chain (DTMC). This framework is general enough to handle the creation of various different filter variants. The filter implementation to be verified is specified in a mainstream programming language (we use Java) since it needs access to linear algebra data types and operations.

Once the DTMC has been constructed, we verify properties related to numerical and modelling errors of the Kalman filter being modelled. These properties are expressed in a reward-based extension [21] of the temporal logic PCTL (probabilistic computation tree logic) [22]. This requires generation of non-trivial reward structures for the DTMC computed using linear algebra computations on the matrices and vectors used in the execution of the Kalman filter implementation. Generating reward structures for the DTMC using linear algebra computations is of more general interest in terms of the applicability of our approach to analyse complex properties of systems via probabilistic model checking.

We have implemented this framework within a software tool called VerFilter, built on top of the probabilistic model checker PRISM. The tool takes the Kalman filter implementation, a description of the system model being estimated and several extra parameters: the maximum time the model will run, the number of intervals the noise distribution will be truncated into, and the numerical precision, in terms of the number of decimal places, to which the floating-point numbers which are used throughout the model will be rounded. The decision to let the user specify these parameters is particularly important in the modelling and verification of stochastic linear dynamical systems, where the states of the model, which consist of floating-point numbers, as well as the labelling of the states, are the result of complex numerical linear algebra operations. Lowering the numerical precision usually means faster execution times at the possible cost of affecting the accuracy of the verification result. This decision is further motivated by the fact that many Kalman filter implementations run on embedded systems with stringent computational requirements [15], and being able to produce performance guarantees is crucial.

We demonstrate the applicability of our approach by verifying four distinct Kalman filter types. This allows us to evaluate the trade-offs of one versus the other. For the system models, we use *kinematic state models*, since they are used extensively in the areas of navigation and tracking [23, 24]. We evaluate our approach with two distinct models. We also show that our approach can successfully analyse a range of useful properties related to modelling and numerical errors in Kalman filters, and we evaluate the scalability and accuracy of the techniques. Overall, 1852 different filter implementations have been verified effectively. We have made the tool, VerFilter, and supporting files for the results publicly available [25].

In summary, the main research question this thesis addresses is "Can we use formal verification to produce rigorous performance guarantees for resource control and estimation mechanisms in cloud computing?". In particular we have developed a framework based on novel quantitative verification methods which can be used for the verification of the automated resource provisioning mechanisms in the cloud. Finally, in the case where the resource provisioning mechanisms are integrated with state estimators, our work can be thought of as adding an extra layer of verification, checking that the automated decision to be taken is correct.

1.1 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 presents background material both on the areas of cloud computing and state estimation. In Chapter 3 we present a review of the related work by identifying the research that has already been conducted, and how the work in this thesis contributes to it. Chapters 4, 5, 6 and 7 contain the main contributions of this thesis. In Chapter 4 we present and evaluate a novel approach based on quantitative verification to produce performance guarantees for cloud-based auto-scaling policies. In Chapter 5, we build upon the work of Chapter 4, and we present a general framework for modelling Kalman filter implementations operating on linear discrete-time stochastic systems, and techniques to systematically construct a Markov model of the filter's operation using truncation and discretisation of the stochastic noise model. Then, we propose verification techniques for properties which relate to numerical stability and modelling error compensation techniques, respectively. Next, in Chapter 6 we provide details on the implementation of this framework as a software, VerFilter. Chapter 7 demonstrates that the novel verification techniques which were presented and implemented in Chapters 5 and 6 can be used for the verification of various types of Kalman filters, successfully. The major contribution of Chapter 7 is that, through an extensive experimental analysis, we show that probabilistic verification can be used to verify Kalman filters operating on linear discrete-time stochastic systems. Finally, Chapter 8 presents the main findings of this thesis, and provides directions for future work.

1.2 Publications

The following peer-reviewed papers were published throughout the course of the doctoral studies.

1. Alexandros Evangelidis, David Parker, and Rami Bahsoon. 2017. Performance Modelling and Verification of Cloud-based Auto-Scaling Policies. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). IEEE Press, Piscataway, NJ, USA, Pages 355-364, ISBN: 978-1-5090-6610-0, DOI: https://doi.org/10.1109/CCGRID.2017.39

2. Alexandros Evangelidis, David Parker, and Rami Bahsoon. 2018. Performance modelling and verification of cloud-based auto-scaling policies. In Future Generation Computer Systems (FGCS), Elsevier, Volume 87, Pages 629-638, ISSN: 0167-739X, DOI: https://doi.org/10.1016/j.future.2017.12.047

3. Alexandros Evangelidis, David Parker. 2019. Quantitative Verification of Numerical Stability for Kalman Filters. In Proceedings of the 23rd International Symposium on Formal Methods (FM), In: ter Beek M., McIver A., Oliveira J. (eds) Formal Methods – The Next 30 Years. FM 2019. Lecture Notes in Computer Science, vol 11800. Springer, Cham, Pages 425-441, ISBN: 978-3-030-30941-1, DOI: https://doi.org/10.1007/978-3-030-30942-8

The first and second publications, [26] and [27], correspond to the contributions which are presented in Chapter 4. The third publication, [28], corresponds to material which is presented in Chapters 5, 6 and 7. Also, an invited journal version of the third publication is currently in preparation.

CHAPTER 2

Background Material

In this chapter, we introduce background material which is required for this thesis. Section 2.1, introduces definitions which are related to cloud computing. Specifically, Section 2.1 starts with defining concrete terms for the cloud actors which exist in a cloud environment, followed by the cloud computing models and closes with an extensive discussion about auto-scaling policies. In Section 2.2, we present an overview of linear stochastic dynamical systems, and we show how the discretisation process can be performed for the models we consider. In the same section we also give the necessary background on the two types of kinematic models we consider. Then, in Section 2.3 we present results from linear algebra and control theory, that are used in the subsequent chapters. Next, Section 2.4 introduces the Kalman filter, Section 2.5 covers the relevant background on square-root filters and Section 2.6 presents the discrete Riccati equation along with important theorems we later make use of when we deal with steady-state filters. Section 2.7 presents relevant background information with respect to probabilistic model checking and the PRISM tool.

2.1 Cloud Computing Definitions

In this section, we describe the definitions relating to cloud computing that are used throughout this thesis, in order to resolve any potential ambiguity.

Service Level Agreement (SLA). An SLA is a legal document that offers, among other things, a level of QoS with respect to the non-functional requirements that are being guaranteed to customers. In the cloud market there is a marked preference among customers for providers who offer SLAs from those who do not. An SLA also allows potential customers to negotiate their non-functional requirements with the SaaS provider and assesses the appropriateness of the SaaS provider to fulfil them. An SLA can consist of multiple objectives (Service Level Objectives or SLOs) and each SLO usually contains a QoS metric which maps to a particular non-functional requirement (e.g. performance) with a value above or below a given threshold with a view to maximising or minimising it respectively [29].

Cloud provider. Manages the hardware resources in the data centre such as servers, racks, physical machines and provides abstractions of those resources usually through virtualisation to cloud users. The cloud provider has complete control over the physical machines and can use strategies such as virtual machine migration to allocate the resources efficiently. The cloud provider has an SLA that it is responsible for satisfying with its tenants.

Cloud user or tenant. An application or a service provider who wishes to take advantage of the infrastructure of the cloud provider and rents those resources, in order to offer highly scalable solutions to its end users/tenants. The cloud user seeks to exploit the elasticity property of the cloud infrastructure. Similar to the cloud provider, the cloud user might offer an SLA to its end users/tenants for the services that are being offered. In our research, this is the type of SLA we take into consideration.

End user or tenant. The end user, who is also considered a tenant [30], is not directly involved in the resource allocation process, but can generate requests and expects that those requests will be satisfied according to the values that have been specified in the SLA (e.g. a request to the server should not take more than 3 seconds to complete).

Software as a Service (SaaS). Refers to on demand software which is offered as a cloud-based service and can be accessed by web browsers. It is important to note that a SaaS provider is not necessarily the cloud provider, and as a matter of fact many service providers have moved their services to the cloud to benefit from economies of scale. This means that service providers choose not to acquire and deploy their services on their own infrastructure (that is on their own compute and storage nodes) and prefer to rent those resources from a cloud provider in order to be able to elastically allocate and de-allocate resources according to the demand.

Platform as a Service (PaaS). Refers to cloud-based services which offer a platform to developers to build and customise their solutions, without however giving them the ability to control the underlying cloud infrastructure, such as operating systems or storage.

Infrastructure as a Service (IaaS). Provides a greater degree of control to the cloud user than the PaaS model, by giving the user the ability to manage the underlying cloud infrastructure, such as operating systems, storage and deployed applications.

Quality of Service (QoS). QoS is used to describe the non-functional requirements of services such as performance, availability, reliability, security and others, in the form of an appropriate metric. For example, one might wish to use response time as a metric to set an expectation about performance, or mean time between failures for reliability etc. Briefly, QoS is a means to quantify the level of a service by considering the appropriate metrics.

Resources and resource provisioning. Resources can be categorised into compute, networking, storage and energy resources. From the cloud provider's perspective the resources above can be managed at the physical level (setting up the physical machines, the servers etc.). In our research, "resources" refer to the virtual infrastructure (whether that is a virtual machine, virtual disks (for storage) or virtual networks) that is being offered to the cloud application provider through virtualisation technology.

From the perspective of a cloud user, resource provisioning refers to the process of efficiently allocating their virtualised resources to its tenants, by looking to meet his/her but also the tenants' objectives. The objectives could range from satisfying the non-functional requirements in the SLA offered, to being able to continuously adapt its services to minimise costs.

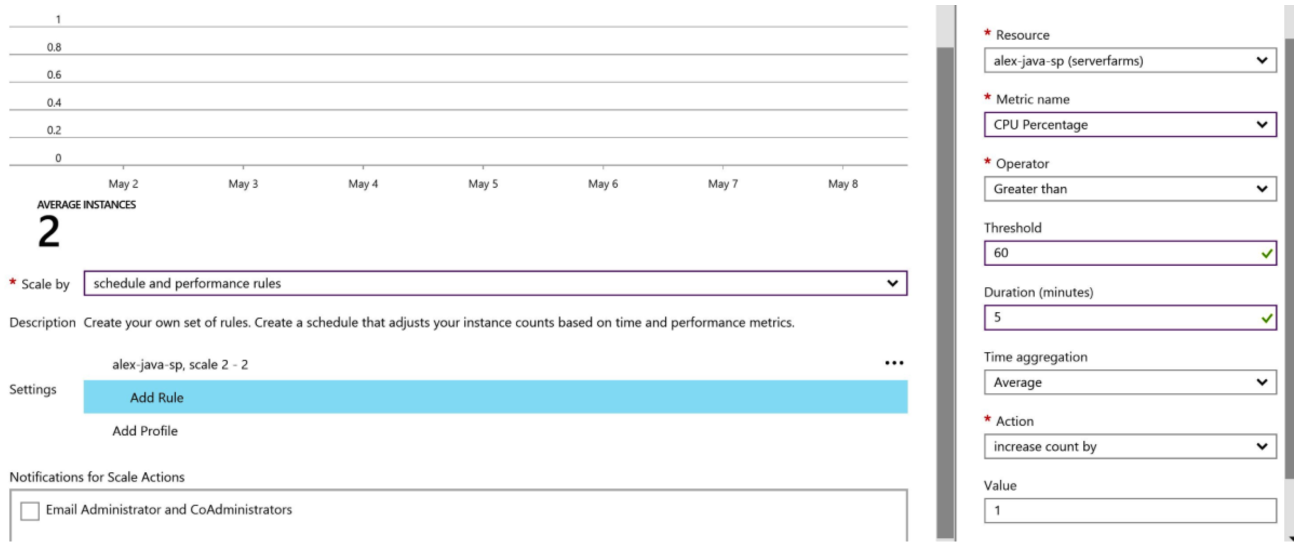
Auto-scaling/Elasticity. Auto-scaling is one of the key properties of cloud computing [31, 32, 33]. In the literature there are a number of definitions regarding elasticity which is often considered as a synonym for scaling. To avoid ambiguity throughout the thesis we adopt the definition of elasticity used in [32]: *"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible"*.

Elasticity is a dynamic property that involves sophisticated concepts such as effective resource management policies and can be examined from two perspectives. The first perspective involves taking the view of the cloud provider who seeks to optimise the resource usage of its infrastructure at the level of racks, servers and Virtual Machines (VMs), in order to minimise the operating costs while offering reliable infrastructure services to its tenants. Tenants in this case refer to the application providers that host their applications in the infrastructure of a cloud provider. The second view, which is the one we adopt in this work, is from the angle of the SaaS provider who wants to optimally allocate and deallocate resources on demand, in order to provide reasonable guarantees to its tenants, in the form of an SLA. Tenants here refer to the end users of the application that is hosted in the cloud. In summary, the core problem around elasticity that refers both to the cloud providers and cloud users, is the minimisation of time the system, whether it is a cloud layer or an application, spends in an under-provisioned or over-provisioned state. Moreover, elasticity considers the temporal aspects of scaling [32] and is strictly related to acceptable QoS criteria, which are specified in SLAs, and can be thought of as a scaling-out and -in method based on the demand that the software is experiencing.

Taking into consideration the above definition, we note that, in contrast to scalability, elasticity is a dynamic property of the application running on the cloud, which considers the temporal aspects of scaling. However, it is worth noting that cloud providers are not willing to take the responsibility for auto-scaling an application deployed by someone else on their infrastructure. Furthermore, cloud providers give the ability to cloud application owners to automate the process of auto-scaling, by configuring their own auto-scaling policies.

An *auto-scaling policy* [7] defines the conditions under which, capacity will be added to or removed from a cloud-based system, in order to satisfy the objectives of the cloud user. Auto-scaling is divided into scaling-up/-down and scaling-out/-in methods, with the two approaches also being defined as *vertical* (add more RAM or CPU to existing VMs) and *horizontal* (add more "cheap" VMs) scaling. In our research, we focus on scaling-out and -in since it is a commonly used and cost-effective approach.

The main auto-scaling method that is given to application providers by all public cloud providers today (e.g. Amazon EC2, Microsoft Azure, Google Cloud) is *rule-based.* The rule-based method is the most popular and is considered to be the state-of-the-art in auto-scaling an application in the cloud [5]. In a rule-based approach, the application provider has to specify an upper and/or lower bound on a performance metric (e.g. CPU utilisation) along with the desired change in capacity



AVERAGE INSTANCES
2

* Scale by: schedule and performance rules

Description: Create your own set of rules. Create a schedule that adjusts your instance counts based on time and performance metrics.

Settings: alex-java-sp, scale 2 - 2

Notifications for Scale Actions

☐ Email Administrator and CoAdministrators

* Resource: alex-java-sp (serverfarms)

* Metric name: CPU Percentage

* Operator: Greater than

Threshold: 60

Duration (minutes): 5

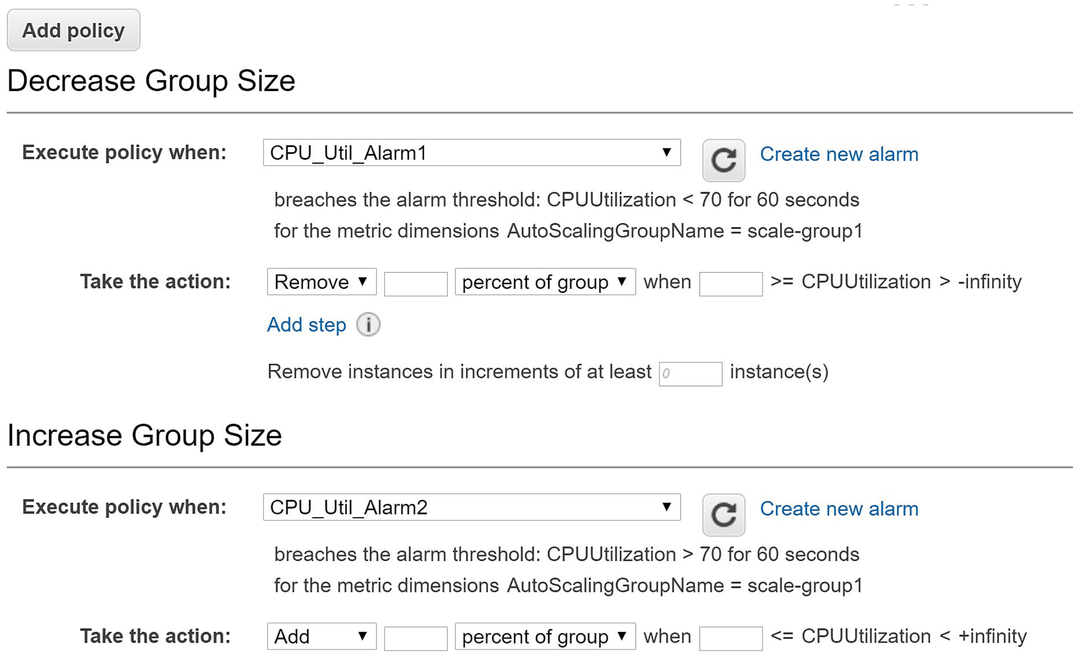
Time aggregation: Average

* Action: increase count by

Value: 1

Figure 2.1: Microsoft Azure's auto-scale control panel.

for this situation. For example, a rule-based method that will trigger a scale-out decision when CPU utilisation exceeds 60% might take the form: *if cpu_utilisation* $> 60\%$ *then add* 1 *instance* [6]. The performance metrics that public cloud providers usually follow include CPU utilisation, throughput and queue length. We consider auto-scaling decisions based on CPU utilisation as it is one of the most important metrics in capacity planning, and also the most widely used in auto-scaling policies. In addition, an auto-scaling policy consists not only of performance metrics thresholds, but also of *temporal* parameters, which often seem to be neglected, despite their significance in configuring a good auto-scaling policy. These parameters include the time interval that the auto-scaling mechanism looks back to determine whether to take an auto-scale action, and the duration for which it is prohibited from triggering auto-scale actions after a successful auto-scale request (cool-down period). Since both of these parameters have to be specified by a human operator, it becomes a challenging task to understand the impact of these parameters on performance metrics of the application running on the cloud. It is exactly this impact that we wish to quantitatively analyse. In Figures 2.1 and 2.2 we show the auto-scale control panel of two major public cloud providers, Microsoft Azure and Amazon EC2, respectively.


Add policy

Decrease Group Size

Execute policy when: CPU_Util_Alarm1 Create new alarm

breaches the alarm threshold: CPUUtilization < 70 for 60 seconds
for the metric dimensions AutoScalingGroupName = scale-group1

Take the action: Remove percent of group when >= CPUUtilization > -infinity

Add step

Remove instances in increments of at least 0 instance(s)

Increase Group Size

Execute policy when: CPU_Util_Alarm2 Create new alarm

breaches the alarm threshold: CPUUtilization > 70 for 60 seconds
for the metric dimensions AutoScalingGroupName = scale-group1

Take the action: Add percent of group when <= CPUUtilization < +infinity

Figure 2.2: Amazon EC2's auto-scale control panel.

2.2 Linear Stochastic Dynamical Systems

The Kalman filter tracks the state of a *linear stochastic dynamical system*, which can be thought of as a system whose state vector evolves over time under the effects of noise. Taking out the word *stochastic* for a moment, linear dynamical systems have gained tremendous popularity in fields ranging from aerospace engineering to economics. Applications include but are not limited to classical mechanics (Newton's laws), population and supply chain dynamics, stock markets and others [34]. As a simple example, we can consider the following dynamical system, where x_{k+1}, the state vector at time step $k+1$, is a *linear* function of x_k, the state vector at time step k, with the F_k matrix denoting the *state transition matrix* or *dynamics matrix* [34].

$$x_{k+1} = F_k x_k \tag{2.1}$$

The model described in the equation above is also called a *Markov model* [34] since the current state x_k contains all the necessary information for x_{k+1} to be calculated.

In estimation problems and especially those related to Kalman filters most of the system models are described by a set of *ordinary* differential equations [14], since most of the models which describe real world phenomena are in continuous time. However, in order for these continuous-time models to be implemented and simulated in "digital circuits" they have to be discretised in order to be transformed to their equivalent discrete-time form [15].

This process is called *discretisation* and can be thought of as a preprocessing step for the Kalman filter. In addition to the discretisation of a continuous model, one also could use *direct* discrete-time models [35]. In our work we focus on kinematic state models and we model them using both discretised and discrete approximations. In general, kinematic state models describe the motion of objects as a function of time, using so-called kinematic equations. These are models which have been used extensively in the areas of navigation and tracking.

In order to demonstrate the discretisation process, let us assume the following *noiseless* kinematic model, a continuous-time linear dynamical system, which is also called an *exact* constant velocity model [23]. This model can be defined as a first order differential equation of the following form:

$$\dot{x} = A(t)x(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x(t) \tag{2.2}$$

In equation 2.2 $\dot{x}$ is defined as $\frac{dx}{dt}$, and this equation represents a continuous-time linear dynamical system in a more compacted form, in the so-called *state space* format. Also note that the state vector $x \in \mathbb{R}^n$, in the two-dimensional case we consider, is given as $x = [p \quad \dot{p}]^T$, where the first element defines the position and the second the velocity. In order to transform equation (2.2) to its discrete counterpart with a sample time of Δt one has to compute the matrix exponential ($e^{A\Delta t}$), often referred to as the *fundamental matrix* Φ in the control theory literature [14]. Here, to avoid confusion with the later parts, we will denote it F_k. Also, note the difference between F_k and $A(t)$: the first denotes discrete time instants k while in the latter case the matrix A is a function of time t, which is continuous.

The matrix exponential can be computed by taking a Taylor series, an infinite series, of the exponential and then substituting the matrix $A\Delta t$.

$$F_k = e^{A\Delta t} = I + A\Delta t + \frac{A^2\Delta t^2}{2!} + ... = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{2.3}$$

It turns out that in our case and for the system models considered, we can obtain an *exact* answer by summing the first two terms of the Taylor series, since A^2 is equal to 0, which means all the subsequent terms of the summation will be 0. As a result, we are able to calculate F by summing over a finite amount of terms from the Taylor series.

Another method we can employ to obtain a closed-form solution of the problem above is to use the *Laplace transform* and then take its *inverse*, providing that the system is *time-invariant* [23, 15]. Time-invariance means that the matrix A is constant ($A(t) = A$) and not a function of time. This is not an unrealistic assumption, since the majority of linear systems are time-invariant [36]. This means that we can also drop the subscript k from F ($F_k = F$) since it will also be constant, and will not depend upon different discretisation step sizes Δt. However, note that, despite the fact that the system is time-invariant, the Kalman filter in most cases is not (it is time-varying). The Laplace transform can be thought of as a general method of taking a function in the time domain such as a differential equation, and transforming it to the frequency domain (also called Laplace domain [37]) where it is easier to solve, since it involves only algebraic operations. Then, once we have the solution in the frequency domain we can transform it back to the time domain, by applying the inverse of Laplace transform. The Laplace transform of a function $f(t)$ can be written in compact notation as $F(s) = \mathcal{L}(f(t))$, where $F(s)$ and $f(t)$ are functions in the frequency and time domain respectively. $F(s)$ can be expanded as follows:

$$F(s) = \int_0^\infty e^{-st} f(t) dt \tag{2.4}$$

In the equation above we can see that, once we integrate over time and apply the limits of integration, the time variable t vanishes and we are left only with s. Analogously, the inverse of a Laplace transform can be written as $f(t) = \mathcal{L}^{-1}(F(s))$, and in this case, since we are dealing with differential equations, the inverse can be written as $\mathcal{L}^{-1}\left((sI - A)^{-1}\right)$ [23]. To obtain the previous equation the derivative property of the Laplace transform is used which is $\mathcal{L}(f'(t) = sF(s) - f(0)$, and is the Laplace transform solution for the continuous time-invariant linear dynamical system defined earlier, $\dot{x} = Ax$ [36]. The computations proceed as follows:
First we transform the equation to the frequency domain by applying the Laplace transform operator:

$$(sI - A) = \begin{bmatrix} s & -1 \\ 0 & s \end{bmatrix} \tag{2.5}$$

Then, we compute the inverse of the matrix above (providing that is invertible), which is the solution in the frequency domain:

$$(sI - A)^{-1} = \begin{bmatrix} \frac{1}{s} & \frac{1}{s^2} \\ 0 & \frac{1}{s} \end{bmatrix} \tag{2.6}$$

Now we apply the inverse Laplace transform to transform the aforementioned solution from the frequency to the time domain:

$$F(\Delta t) = \mathcal{L}^{-1}\left(\begin{bmatrix} \frac{1}{s} & \frac{1}{s^2} \\ 0 & \frac{1}{s} \end{bmatrix}\right) = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{2.7}$$

The solution is what we would expect and identical to the one obtained by summing the Taylor series. For a critical review of the many methods which can be used to compute the matrix exponential, we refer the interested reader to the excellent paper of [38].

2.2.1 Kinematic Models

In our work we focus on two types of kinematic models, which can be broadly classified into: *discretised continuous-time kinematic models* and *direct discrete-time kinematic models.* It is worth noting that information in the literature about kinematic models is not presented in a unified and systematic manner. This issue was reported by [39] who noted the *"scatteredness"* and *"unavailability"* of the various motion models which had been developed from 1970 onwards, and made an attempt to summarise them. Later, the same authors refined their original paper and published a series of papers in separate parts (part I to V) related to target tracking. Part I of this series [24] is an excellent survey on dynamic models and motion models in particular.

Until that point, one of the notable collective works which surveyed motion models specifically was from [40], whose book was, however, criticised by [39] as "far from complete". Of course, many models in this book were based upon previous works of [41] and others, mostly among the radar tracking literature. Later, some of the previous authors collaborated and wrote a book [23], which is considered one of the standard references in the tracking and navigation literature. In fact, many of high quality papers in this area use the results of the aforementioned book to justify the values for their motion model parameters. Similar to those, the choice of kinematic models which are described here are based upon this book.

Discretised Continuous White Noise Acceleration Model (CWNA). In general, kinematic models describe the motion of objects (e.g. physical systems) as a function of time. In particular, the CWNA model assumes that the object's velocity is perturbed randomly by *continuous time white noise.* The equation of the model in continuous-time is given by $\dot{x} = Ax(t) + D\tilde{w}(t)$ where A is the state transition matrix defined in (2.2), and D is defined as the *noise gain* or the noise distribution matrix for the system noise $\tilde{w}(t)$ [23, 42].

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad D = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.8}$$

The continuous-time white noise is a stochastic process $\tilde{w}(t)$ whose mean is defined as $E[\tilde{w}(t)] = 0$ with autocovariance $E[\tilde{w}(t)\tilde{w}(\tau)] = q(t)\delta(t - \tau)$, where τ is the amount of time by which the signal has been shifted, δ is the Dirac delta function, and q is the power spectral density [13]. Note that *white noise* does not exist in the real world; mathematically it could mean a process with infinite variance [43]. However, white noise is a very useful model to describe the various random effects on a system in different scientific fields. Formally, the power spectral density of a wide sense stationary (WSS) process is defined as the *discrete-time Fourier transform* (DTFT) of its autocorrelation [44], and in the case of white noise, the power spectral density is constant. Note that the Fourier transform is used to transform the autocorrelation function, which is expressed in the time domain, to the power spectral density, which is a function defined in the frequency domain. It can be shown that the power spectral density of a white noise process is equal to its variance (σ_w^2). This is an important mathematical result which of course stems from other important theorems (e.g. Wiener-Khinchin theorem) in the signal processing literature. For a derivation of the above result, the interested reader can find more information in [44].

The discretised model of the above system, assuming that we sample it at discrete-time intervals Δt, is given as $x_k = Fx_{k-1} + w$, where the state vector x_k is a linear function of the initial state plus the additive noise. The covariance noise matrix of w, Q, is computed in a relatively similar manner to the state transition matrix F, and is given as:

$$Q = \begin{bmatrix} \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 \\ \frac{1}{2}\Delta t^2 & \Delta t \end{bmatrix} q \tag{2.9}$$

where q is the power spectral density of the noise defined previously. Note that we drop the subscript k from the Q matrix since we treat the noise process w as a *stationary* process which means that its mean and covariance will remain constant over time.

Discrete White Noise Acceleration Model (DWNA). The DWNA model, also called *piecewise constant* acceleration model [23], assumes that the acceleration remains constant for each time interval Δt. The equation of the model is given by the following discrete-time equation: $x_{k+1} = Fx_k + \Gamma w_k$, where F is the state transition matrix, and Γ is defined as the noise gain matrix [45]. F and Γ are given as:

$$F = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \qquad \Gamma = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix} \tag{2.10}$$

Note that with the choice of the above noise gain Γ we compute the updated position in one interval Δt as $w_k\frac{1}{2}\Delta t$, and the updated velocity as $w_k\Delta t$. In other words we can think of w_k as the velocity which undergoes slight changes in different time steps. Finally, the covariance noise matrix Q is:

$$Q = \Gamma\sigma_w^2\Gamma^T = \begin{bmatrix} \frac{1}{4}\Delta t^4 & \frac{1}{2}\Delta t^3 \\ \frac{1}{2}\Delta t^3 & \Delta t^2 \end{bmatrix} \sigma_w^2 \tag{2.11}$$

Note that σ_w^2 is the variance of the white noise, and is a scalar similar to the q variable from the CWNA model.

2.3 Linear Algebra and Control Theory Preliminaries

In this section, we aim to give an overview of some important mathematical characteristics which are going to be needed in the subsequent sections. A symmetric matrix is a square matrix equal to its transpose ($A = A^T$). The symmetric property of a covariance matrix can be derived from the definition of covariance, and holds for every covariance matrix. The *positive definiteness* property of a symmetric matrix is usually more complicated since it can be defined by several equivalent mathematical statements. A symmetric positive definite matrix has eigenvalues which are *real* and positive, which implies that the matrix is *invertible* or *nonsingular* [15]. Note that it is because of the symmetric property that the eigenvalues are real.

Eigenvectors have played a prominent role in control theory to analyse systems and their properties, from different perspectives. For example, in many cases it is considered easier and more efficient to analyse the dynamics of the system under consideration by transforming it from its initial coordinates to some eigenvector coordinates, usually in the complex plane. Formally, the analysis of the dynamics of a system can be compressed into the analysis of the following three properties: *stability*, *observability* and *controllability*, and especially the last two are considered “fundamental in modern control theory” [15, 46]. Also, these properties are defined differently for continuous-time and discrete-time systems, and their definition depends upon whether the system is time-varying or time-invariant. In our work, we focus on linear, discrete-time, time-invariant systems and our discussion is centred around these types of systems only.

Stability answers the question of how well the system behaves as time goes to infinity, which practically means that x_k converges to 0 as $k \to \infty$ [36]. A system is stable if the eigenvalues of the state transition matrix F have magnitude less than one. The notion of stability is related to the Kalman filter and more concretely to the propagation of the a priori estimation-error covariance matrix, the so-called

Lyapunov equation (see equation 2.30). In particular, if F_k is *stable* and constant, and Q_k is constant, the Lyapunov equation has a unique, symmetric *steady-state solution.* In other words, the *steady-state a priori covariance* of the estimation error can be computed by solving this equation [15, 36].

To describe observability, we first have to define the so-called measurement equation which is associated with a linear dynamical system, such as, for example, the one in equation 2.1. This equation describes the observation process at discrete time instants k, and is given as $z_k = H_k x_k$ where z_k is the measurement vector and H_k is the measurement matrix which relates the measurement with the state vector x_k.

Observability, is concerned with being able to estimate the state x at any given time k from the measurements z. A common test for observability is to check whether the *rank* (e.g. number of linearly independent rows/columns) of the so-called observability matrix is equal to the dimension n of the state vector x. The observability matrix M can be constructed as follows [14]:

$$M = \begin{bmatrix} H^T & F^T H^T & (F^T)^2 H^T & \dots & (F^T)^{n-1} H^T \end{bmatrix} \tag{2.12}$$

Additionally, observability can be defined in terms of the Lyapunov equation expressed earlier; for a compact representation of those theorems, we refer the interested reader to [15]. As discussed earlier, the system is observable if and only if $\rho(M) = n$, where ρ is the rank of matrix M.

Controllability generally refers to the idea of being able to control the system by using some form of feedback. More precisely, a system is controllable if the elements of the state vector x can be affected by the control input [47]. In our case, the control input is the process noise w of the DWNA model in Section 2.2.1. The controllability test resembles the observability test, only in this case the rank of the controllability matrix is examined. The controllability matrix S can be constructed as follows [14]:

$$S = \begin{bmatrix} \Gamma & F\Gamma & F^2\Gamma & \dots & F^{n-1}\Gamma \end{bmatrix} \tag{2.13}$$

The system is controllable if and only if $\rho(S) = n$, where ρ is the rank of matrix S and n the dimension of state vector x. The notion of controllability is very important because it can be extended and applied even if the system of interest has no input gain matrix and vector.

2.4 The Kalman Filter

One of the most popular algorithms for estimating the state of a continuously changing system based on uncertain information about its dynamics is the Kalman filter. Despite the fact that there exists a vast body of literature on the Kalman filter, understanding it in-depth can be difficult. This is mainly because the Kalman filter can be examined from different viewpoints, and sometimes this can cause confusion to the reader. For example, in the control theory literature it is often referred to as a *linear quadratic estimator* (LQE) [48] where it is combined with a state feedback controller, usually under the name *linear quadratic regulator* (LQR) to control a dynamical system. Others may refer to it as a specialised instance of the *Bayes filter* [49], where it can be seen as a recursive Bayesian estimation algorithm for Gaussian distributions. Other authors describe it as a *linear mean squared error estimator* [14], others as a "*linear, finite-dimensional system*" [35] or simply as a *conditional mean estimator* [50].

What is "*remarkable*"[15], however, is that these different viewpoints are interlinked and can lead to the same conclusion: the Kalman filter is the best minimum mean squared error estimator among all estimators (linear and nonlinear) under the Gaussian assumption [14, 23, 13, 15, 51]. Even if the Gaussian assumption is relaxed, the Kalman filter is still the best *linear* minimum mean squared error estimator among all estimators in the linear class.

The Kalman filter tracks the state of a linear stochastic discrete-time system of the following form:

$$x_{k+1} = F_k x_k + w_k \tag{2.14}$$

$$z_k = H_k x_k + v_k \tag{2.15}$$

where x_k is the (n $\times$ 1) system state vector at discrete time instant k, F_k is a square (n $\times$ n) state transition matrix, which relates the system state vector x_k between successive time steps, in the absence of noise. In addition, z_k is the (m $\times$ 1) measurement vector, H_k is the (m $\times$ n) measurement matrix, which relates the measurement with the state vector. Finally, w_k and v_k represent the process and measurement noises, with covariance matrices Q_k and R_k, respectively. Given the above system and under the assumption that both the system's and measurement's noises are Gaussian, zero mean and uncorrelated, the Kalman filter is an optimal estimator in terms of minimising the mean squared estimation error. Specifically, the covariance matrices of w_k and v_k are:

$$p(w_k) \sim \mathcal{N}(0, Q_k) \qquad \mathbb{E}[w_k w_i^T] = \begin{cases} Q_k, & i = k \\ 0, & i \neq k \end{cases} \tag{2.16}$$

$$p(v_k) \sim \mathcal{N}(0, R_k) \qquad \mathbb{E}[v_k v_i^T] = \begin{cases} R_k, & i = k \\ 0, & i \neq k \end{cases} \tag{2.17}$$

$$\mathbb{E}[w_k v_i^T] = 0 \tag{2.18}$$

The Gaussian assumption associated with the state vector x_k and measurement vector z_k is particularly important because any linear combination of Gaussian random variables preserves their Gaussian properties [35, 23]. Another benefit this assumption provides is that a Gaussian distribution can be characterised only by its first and second moments (mean and covariance) [45, 13]. This justifies the efficiency of the Kalman filter since, from a Bayesian viewpoint, it can propagate conditional density functions forward in time using only those two pieces of information. Finally, as we will see later, the Gaussian assumption can provide us with some worst-case guarantees since, under this assumption, the *theoretical* performance of the filter can be defined in terms of the estimation-error covariance matrix P_k [23, 14].

Before we give an overview of the estimation process, it is worthwhile to explain the notation that will be used, by distinguishing two types of estimates that are being used. The $\hat{x}_k^-$ notation stands for the *a priori* state estimate at time step k, with the "hat" symbol denoting the estimate, and the minus superscript denoting that the measurements at time k have not been processed yet. Mathematically, it can be written as $\mathbb{E}[x_k \mid z_1, z_2, z_3, ...z_{k-1}]$, which is the conditional expectation of the random variable X_k given the measurements up to and including the time step at $k-1$. Analogously, the $\hat{x}_k^+$ denotes the *a posteriori* state estimate at time k, meaning that in this case the measurements at time step k have been taken into account in the estimation of x_k, and can be written as $\mathbb{E}[x_k \mid z_1, z_2, z_3, ...z_k]$. Each of those estimates, which are essentially estimates of the same variable under different time steps, have their associated *a priori*, and *a posteriori* estimation-error covariance matrices which denote the uncertainty associated with the respective state estimates. The *a priori* estimation-error covariance matrix P_k^- of the *a priori* state estimate $\hat{x}_k^-$ and the *a posteriori* estimation-error covariance matrix P_k^+ of the *a posteriori* state estimate $\hat{x}_k^+$ can be computed as follows:

$$P_k^- = \mathbb{E}[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^T] \tag{2.19}$$

$$P_k^+ = \mathbb{E}[(x_k - \hat{x}_k^+)(x_k - \hat{x}_k^+)^T] \tag{2.20}$$

The quantity $x_k - \hat{x}_k$ is the *estimation error* and the optimality criterion is defined in terms of minimising the variance or the mean squared error of this quantity, since $\mathbb{E}[(x_k - \hat{x}_k)^2] = \text{Var}(x_k)$, in the case there is no bias in the estimates. The reason the performance criterion has been expressed in two equivalent terms is that it can help us to develop our intuition with respect to how the estimation procedure in a Kalman filter works, by drawing analogies from the least squares and Bayesian estimation theories where needed. Moreover, in order to justify why the expected value of x_k is considered the optimal estimate it might be useful to consider the least mean squares estimate in the special case where there are no observations available. Recall that the mean squared error (MSE) is defined as $\mathbb{E}[(X - \hat{x})^2]$ where X is a random variable we are trying to estimate and $\hat{x}$ is the estimate. The fact that we treat the unknown variable X as random means that we take a Bayesian view in our reasoning, which implies that X has an associated prior distribution to it. Also, for convenience we have dropped the subscripts k. The MSE can be expanded as follows:

$$\mathbb{E}[(X - \hat{x})^2] = \mathbb{E}[X^2] - 2\mathbb{E}[X] + 2\hat{x}^2 \tag{2.21}$$

Then, in order to find the value which minimises the MSE, we differentiate equation 2.21 with respect to $\hat{x}$, and set its derivative to zero.

$$\frac{d}{d\hat{x}} = 0 : \tag{2.22}$$

$$-2\mathbb{E}[X] + 2\hat{x} = 0 \tag{2.23}$$

$$\hat{x} = \mathbb{E}[X] \tag{2.24}$$

From the above equation we can observe that the MSE is minimised when $\hat{x} = \mathbb{E}[X]$, and in that case the optimal value of the mean squared error is the variance of X [52]. The case above is identical to the case of no observations. Assuming the existence of incoming measurements, the optimality criterion is defined in terms of minimising the *conditional* mean squared error given by $\mathbb{E}[(X - \hat{x})^2 \mid Z = z]$. This estimator is defined as *unbiased*, and is the *conditional mean* or the minimum mean squared error (MMSE) estimator. It has an important uniqueness property, meaning that no other estimator can perform better in terms of minimising the mean squared estimation error [52, 23].

In order to proceed with the derivation of the Kalman filter it is convenient to express the computation of the a posteriori state estimate recursively, which is similar to the *recursive least squares* (RLS) estimation algorithm expressed in the following linear form:

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(z_k - H_x\hat{x}_k^-) \tag{2.25}$$

The K_k term is called the *Kalman gain* matrix and will be derived shortly. In the scalar case it can be thought of as a weighting factor whose entries take values in the interval [0..1] and adjusts the a priori state estimate according to how much "trust"/ "belief" is placed on the newly obtained measurements. For example, if the Kalman gain is zero that would mean that there is no uncertainty associated with the a priori estimate $\hat{x}_k^-$, and as a result the a posteriori state estimate $\hat{x}_k^+$ would equal the a priori state estimate $\hat{x}_k^-$. The expression in the parenthesis is the *residual* or the *innovation*, which essentially is the difference between the measurement obtained at time k (z_k) and the a priori state estimate $\hat{x}_k^-$. It is important to note, however, that in the RLS algorithm the vector $\hat{x}$ is treated as a constant. Many of the standard books related to estimation make this distinction, and they prefer to call this recursive estimation process *"dynamic estimation as a recursive static estimation"* [23] and others as some form of *"updating least squares"* [53] where the estimate is updated as new measurements are being obtained.

The task of the Kalman filter is to find the optimal Kalman gain matrix in terms of minimising the sum of estimation error variances or the mean squared estimation error. The sum of the variances of the estimation error can be obtained by summing the elements of the main diagonal (trace) of the a posteriori estimation-error covariance matrix P_k^+. After making the necessary substitutions in equations 2.19 and 2.20, P_k^+ can be expressed in the following two equivalent forms:

$$P_k^+ = (I - K_k H_k) P_k^- (I - K_k H_k)^T + K_k R_k K_k^T \tag{2.26}$$

$$P_k^+ = (I - K_k H_k) P_k^- \tag{2.27}$$

Now, in order to solve for the optimal Kalman gain at time k, we differentiate the trace of P_k^+ with respect to K_k and then set its derivative equal to zero, to obtain the following:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{2.28}$$

In the Kalman gain equation 2.28 the computation inside the parenthesis is the covariance of the innovation and usually it is calculated as a separate step before the Kalman gain computation. The innovation covariance matrix is usually referred to in the literature as S.

The estimation process begins by initialising $x_0^+ = \mathbb{E}[x_0]$, and $P_0^+ = \mathbb{E}[(x_0 - \hat{x}_0^+)(x_0 - \hat{x}_0^+)^T]$. Then, the way *linear difference equation* propagates the state estimate of x_k forward in time is given by taking the expected value of both sides of the equation 2.14, which results in $\hat{x}_1^- = F_0 \hat{x}_0^+$ or the more general version:

$$\hat{x}_k^- = F_{k-1} \hat{x}_{k-1}^+ \tag{2.29}$$

The a priori estimation-error covariance matrix P_k^- is propagated in a similar manner and its equation, which is known in the control theory literature as the *discrete-time*

Lyapunov equation, is the following [15]:

$$P_k^- = F_{k-1} P_{k-1}^+ F_{k-1}^T + Q_{k-1} \tag{2.30}$$

Then, the Kalman filter proceeds by iterating between two steps. The first step is called the prediction step or time update, while the second is the correction step or the measurement update. The conventional Kalman filter algorithm is summarised below:

Time update:

$\hat{x}_k^- = F_{k-1} \hat{x}_{k-1}^+$

$P_k^- = F_{k-1} P_{k-1}^+ F_{k-1}^T + Q_{k-1}$

Measurement update:

$y_k = z_k - H_k \hat{x}_k^-$

$S_k = H_k P_k^- H_k^T + R_k$

$K_k = P_k^- H_k^T S_k^{-1}$

$\hat{x}_k^+ = \hat{x}_k^- + K_k y_k$

$P_k^+ = (I - K_k H_k) P_k^- (I - K_k H_k)^T + K_k R_k K_k^T$

or

$P_k^+ = P_k^- - K_k S_k K_k^T$

or

$P_k^+ = (I - K_k H_k) P_k^-$

2.5 Square-Root Filtering

Square-root filters are generally considered superior to conventional filter implementations mainly because of their ability to increase the numerical stability of the propagation of the estimation-error covariance matrix P, and have often been described as outstanding [13, 54]. It should be noted that the term square-root filter is mostly used to refer to the measurement update of the Kalman filter algorithm, since it is this part that can cause numerical problems [16]. They were motivated by the need for increased numerical precision because of word lengths of limited size in the 1960s [15] and by the concern with respect to the numerical accuracy of P in the measurement update of the Kalman filter equations [16]. Potter [55] proposed the idea of the so-called square-root filters and this idea has evolved ever since. The idea, which was limited to noiseless systems, is that P is factored into its square root C, such that $P = CC^T$, and as a result C is propagated through the time and measurement update equations, instead of P. This means that replacing P with its square-root factor C has the effect of doubling the numerical precision of the filter, thus making it particularly suitable for matrices which are not well-conditioned or when increased precision cannot be obtained from the hardware [14, 13, 15, 16].

2.5.1 The Carlson-Schmidt Square-Root Filter

The Carlson-Schmidt filter is a form of a square-root filter which relies on the decomposition of P into its Cholesky factors in the time and measurement update equations. The Carlson part of the filtering algorithm, originally given by Carlson [56], corresponds to the measurement update, while the Schmidt part corresponds to the time update of the Kalman filter equations, respectively. Carlson's algorithm is capable of handling noise and, like Potter's algorithm, processes measurements as scalars. It factors P into the product of an upper-triangular Cholesky factor and its transpose such that $P = CC^T$. Note that, unlike Potter's initial square-root filter where the factor C is not required to be triangular, in Carlson's square-root implementation the Cholesky factor C is an upper-triangular matrix. Maintaining C in upper-triangular form has been shown to provide several advantages in terms of storage and computational speed compared to Potter's algorithm [13, 56, 57]. While the choice between a lower and upper-triangular Cholesky factor C is arbitrary [13], Carlson motivated the preference to choose an upper-triangular Cholesky factor by the fact that in the time update part of the algorithm, fewer retriangularisation operations are required especially when someone designs a filter to be applied in a tracking or in a navigation problem, respectively [56].

2.5.2 The Bierman-Thornton U-D Filter

The Bierman-Thornton filter, or U-D filter for short, is one of the most widely used Kalman filter variants [58], which despite its appearance in the early 1970's, due to its numerical accuracy, stability and computational efficiency it is *"still the dominant type of factored filter algorithm"* [16]. It is worth noting that in the literature there seems to be some ambiguity as to whether the U-D filter is considered a square-root filter or not, since there are authors who classify it under the broader category of square-root filters and others who do not [14, 15]. Strictly speaking, the U-D filter is not a square-root filter and therefore some authors use the term "factored filter" [16] to refer to it. Specifically, the "Bierman" part of the filtering algorithm, originally given by Bierman [59] corresponds to the observational update, while the "Thornton" part given by Thornton [57] corresponds to the time update of the Kalman filter equations, respectively.

Bierman's covariance update, the "actual" U-D filter relies on the decomposition of P into the following matrix product: $P = UDU^T$, where U is a unit upper-triangular and D is a diagonal matrix, respectively [60], a procedure which is often referred to as a modified Cholesky decomposition and the U, D factors as modified Cholesky factors [14]. Unlike Carlson's method it does not require computing scalar square roots for every incorporated measurement [13, 60, 57], thus making it rather suitable for problems where the number of variables defining the state space is large [14]. Furthermore, Bierman's algorithm in a manner similar to Carlson's method promotes the use of upper-triangular matrices for the same reasons of computational efficiency. Thornton's algorithm provides an alternative for the conventional Kalman filter's time-update equations as it propagates the U and D factors, instead of P, forward in time, using the numerically stable Modified Weighted Gram-Schmidt (MWGS) orthogonalisation algorithm [57].

2.6 The Discrete Algebraic Riccati Equation

As it will become evident in the upcoming sections, the *discrete-time matrix Riccati equation*[1] or in other words the propagation of the estimation-error covariance matrix P_k in each time step, is so important in the Kalman filter that it deserves a separate section. The discrete-time matrix Riccati equation or just the Riccati equation for convenience can be formed by combining the a priori and a posteriori covariance equations [23]. The Riccati equation is given by:

$$P_{k+1}^- = F_k[P_k^- - P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} H_k P_k^-] F_k^T + Q_k \tag{2.31}$$

This equation is called a *difference equation*, the discrete version of a differential equation, which describes how the covariance of the estimation error evolves over time. It can also be thought of as a *recursive* equation since the computation of the a priori estimation-error covariance for the next time step (e.g. P_{k+1}^-) depends on the a priori estimation-error covariance of the previous time step (e.g. P_k^-). In fact, in the literature it is often referred to as the Riccati recursion [61], which aligns perfectly with the recursive nature of the Kalman filter. Note that by expressing the a priori estimation-error covariance with the Riccati equation we have eliminated

[1]Named after Jacopo Francesco Riccati (1676-1754)

an extra computation step; the computation of the a posteriori estimation-error covariance matrix P_k^+. Also, P_{k+1}^- is a symmetric, positive semidefinite matrix, which means its eigenvalues ≥ 0. The interesting question is what happens in the limiting case as $k \to \infty$. Under the assumption that the solution in the limit exists, then $P_{k+1}^- = P_k^- \triangleq P$, which leads us to the *discrete algebraic Riccati equation* (DARE) given as:

$$P_\infty = F[P_\infty - P_\infty H^T (H P_\infty H^T + R_k)^{-1} H P_\infty] F^T + Q \tag{2.32}$$

When we seek solutions for the equation we want to limit the choice of possible solutions to those which are positive semidefinite, since P is a covariance matrix. In addition, we are interested in finding the conditions for which these solutions exist. The analysis of Riccati equations from several perspectives is an entire field of mathematical study; here we summarise some of the important theorems which can be found in several books such as [62, 23, 16, 15, 13] which can help us answer our questions.

Theorem 1. *A bounded solution (P) in the limit exists for every P_0, if the matrix pair $\{F, H\}$ is completely observable, and this solution P is positive semidefinite.*

This theorem, despite its importance since it precludes solutions which are negative semidefinite, negative definite and indefinite, which would have been invalid for a covariance matrix, could still lead to a not necessarily unique solution, since the resulting matrix could be either positive definite or positive semidefinite. This rather subtle point is very important for the propagation of the covariance of the estimation error in the Kalman filter. This is because a change in the sign of the eigenvalues of the estimation-error covariance matrix could determine whether the Kalman filter will converge or not. For example a positive semidefinite (e.g. eigenvalues ≥ 0) estimation-error covariance matrix P, which is a valid covariance matrix nonetheless, would mean that the Kalman filter "completely trusts" the estimates for some states of the state vector, and as a result would not "trust" the measurements. Therefore, we are interested in finding those properties that could lead to a *unique positive definite* solution, which leads us to the second theorem [23].

Theorem 2. *Let C be the Cholesky factor, a square root of Q ($\sqrt{Q}$), of the process noise covariance matrix Q, such that $Q = CC^T$. If and only if the pair $\{F, C\}$ is completely controllable then there exists a unique positive definite limiting solution P and this solution is independent of the initial condition P_0.*

The reason this controllability test is performed is to ensure that the process noise will excite every component of the state vector, in order to prevent the estimation-error covariance matrix P from becoming 0.

In the literature one can find proposals about numerical algorithms for solving the Riccati for both the continuous and discrete-time case. However, the study of the Riccati equations is a deep mathematical topic and one should evaluate the various proposals in the literature carefully. It is no accident that state-of-the-art commercial tools, such as MATLAB, rely on techniques which were proposed in the eighties to solve the Riccati equations. In many cases the results for the continuous case can be extended to the discrete one [14]. The classical methods for solving the Riccati equation can be broadly classified into invariant subspaces methods, iterative methods and methods based on the matrix sign function [63]. The most well-known from the matrix sign function methods is the one from [64]. From the iterative methods the most well-known includes the work of Kleinman who was one of the first to use the Newton's method to propose a numerical algorithm for the continuous case [65], in what became known as the Newton-Kleinman iteration. Later, Hewer [66] proposed an iterative method which is the analogue of the Newton-Kleinman method for the discrete-time case. From the invariant subspace methods, and in particular those based on Schur methods, one of the most important is considered the algorithm given initially by Laub [67]. This was later extended with other techniques by Arnold and Laub [68] to produce the numerical algorithmic library RICPACK. RICPACK, which was written in Fortran, was used for solving the Riccati equation amongst other things, and later became part of the popular linear algebra package LAPACK [16, 63, 68]. The techniques proposed in their paper [68] are used today for solving the continuous and discrete-time equations in MATLAB's Control System Toolbox.

2.7 Probabilistic Model Checking and PRISM

Probabilistic model checking is an automatic *quantitative verification* technique which seeks to establish *quantitative* properties which relate to the specification of a probabilistic system, with some degree of mathematical certainty [69, 70]. In order to perform probabilistic model checking two inputs are required: i) a probabilistic model, which is a representation of a probabilistic system and ii) a specification, usually expressed in probabilistic temporal logic [71]. Therefore, *quantitative verification*, and probabilistic model checking in particular, can be thought of as a generalisation of conventional model checking techniques [69, 72].

PRISM [19] is a *probabilistic model checker*, which supports the construction and formal quantitative analysis of various probabilistic models, including discrete-time Markov chains, continuous-time Markov chains and Markov decision processes. These models can be specified in several ways, in particular:

- using PRISM's modelling language, which is how we define our PRISM model related to cloud-based systems in Chapter 4.
- programmatically, using the `ModelGenerator` API, which is how we construct models related to the verification of Kalman filters in Chapter 6.

In our research, for the verification of both auto-scaling policies and Kalman filters, we use discrete-time Markov chains, which are well suited to modelling systems whose states evolve probabilistically, but without any nondeterminism or external control. They are therefore appropriate here, where we want to verify auto-scaling policies and Kalman filter executions, whose outcomes are probabilistic. Formally, a discrete-time Markov chain is defined as follows.

Definition 1. A *discrete-time Markov chain* is a tuple $M = \langle S,P,AP,L \rangle$where:

- S is a finite set of states;
- $\mathbf{P} : S \times S \to [0,1]$ is a transition probability matrix;
- AP is a finite set of atomic propositions;
- L: $S \to 2^{AP}$ is a labelling function.

Each element of the transition probability matrix $\mathbf{P}(s,s')$, gives the probability of transitioning from state s to s'. Since we are dealing with probabilities, we require that $\sum_{s' \in S} \mathbf{P}(s,s') = 1$. If we denote the state of the Markov chain at a particular time step k by X_k, then the transition probabilities can be defined mathematically as $Pr(X_{k+1} = s' | X_k = s) = \mathbf{P}(s,s')$ for any $s, s' \in S$. The set of atomic propositions AP describes properties of interest which can be either true or false in the Markov chain's states, and the labelling function L maps states to the atomic propositions in the set AP.

For discrete-time Markov chains, properties of the model are specified in PRISM using an extension [21] of the temporal logic PCTL (probabilistic computation tree logic) [22]. A typical property is $\mathtt{P}_{\bowtie p}[\psi]$, where $\bowtie \in \{\leq, <, >, \geq\}$ and $p \in [0,1]$, which asserts that the probability of event ψ occurring meets the bound $\bowtie p$. Events are specified using temporal operators, e.g. $\mathtt{F}\Phi$ means that along a path state formula Φ *eventually* holds and $\mathtt{G}\Phi$ means that formula Φ *always* holds in every state of the path. As an example, in our model, where we want to verify whether the auto-scaling decisions will drive the cloud application to a state where the utilisation will be less than 95% with probability greater than 0.7, the following formula will be checked : $\mathtt{P}_{>0.7}[\mathtt{F}(util < 95)]$.

The states of probabilistic models can also be annotated with *rewards* (or costs). A *reward structure* is a labelling function which assigns a nonnegative rational value to a state, and model checking a reward-based property usually refers to the computation of its expected value [71]. Such properties are specified using the reward operator $\mathtt{R}$. For example, a state s satisfies a reward-based property of the form $\mathtt{R}_{\bowtie r}[\mathtt{C}^{\leq \mathtt{k}}]$ if from state s, the expected reward *cumulated* after k time steps satisfies the bound r. Moreover, a state s satisfies a property of the form $\mathtt{R}_{\bowtie r}[\mathtt{I}^{=\mathtt{k}}]$ if, from state s, the expected *instantaneous* reward at time step k meets the bound r. Furthermore, the property $\mathtt{R}_{\bowtie r}[\mathtt{F}\Phi]$ is true in a state s, if the expected cumulated reward before reaching a state in which the formula Φ holds, meets the bound r [71]. Moreover, when a model has multiple rewards structures, the $\mathtt{R}$ operator can be annotated to indicate which one is being used.

In addition, PRISM supports numerical properties such as $\mathtt{P}_{=?}[\mathtt{F}\ \mathit{fail}]$, which means "what is the probability for the cloud application to end up in a failed state, as a result of the auto-scaling decisions made?". Of course, what is considered a *failed* state will differ between cloud application owners, according to the relative importance they put on the non-functional aspects of their application. PRISM allows a wide range of such properties to be specified and analysed. See e.g. [21] for full details of the syntax and semantics of the PRISM property specification language.

CHAPTER 3

Related Work

In this chapter we review the relevant work from the literature, which is closely related to the subject of the thesis: the verified estimation and control for cloud computing, with an emphasis on the automated resource provisioning mechanisms, which are an integral part of today's cloud-based systems. We begin, in Section 3.1, by conducting a thorough literature review on the types of resource provisioning which manifest themselves in cloud computing. Next, Section 3.2 reviews the testing-based approaches for evaluating non-functional requirements in cloud computing and their limitations. Following this, in Section 3.3, we focus on the verification-based techniques for non-functional requirements in the cloud, followed by a discussion of the use of probabilistic model checking in the context of cloud computing in particular. Then, in Section 3.4 we review the use of Kalman filters in resource provisioning contexts. Finally, Section 3.5 contains an extensive discussion on the causes of divergence of Kalman filters, followed by a review on the two most common causes of divergence: modelling and numerical errors.

3.1 Resource Provisioning in Cloud Computing

In the software engineering community, there has been a significant amount of work in proposing methods for optimal resource provisioning in a cloud environment. The resource provisioning methods are divided into two types: static and dynamic approaches. Static provisioning is regarded as the simplest possible solution, since it assumes that the workload patterns will remain unchanged, and as a result the end user will be satisfied with a fixed amount of resources. Furthermore, in cases where it assumes workload variations, the provisioning of resources is based on peak demand therefore not taking into account the cost incurred from the provider's perspective, resulting in a method that is not cost-effective [73].

Dynamic provisioning, on the other hand, is a more challenging problem since it does take into account the fluctuating demand and workload, and is concerned with allocating and de-allocating resources in an efficient manner. Dynamic provisioning is based on the elasticity property of the cloud which has been explained in Section 2.1. Here, and in line with our research, we focus on the dynamic provisioning of resources which is the harder problem and which has attracted the interest of many researchers in the field. Over the past years a lot of work has been conducted in the area of dynamic resource provisioning [74, 75, 33, 76, 77, 78, 79, 80], from a cloud provider/IaaS perspective. Market-based approaches such as [74, 79, 75] view the cloud as a market and focus on adaptive resource provisioning with the goal of maximising revenue for the cloud provider.

Other work such as [81] is based on using Fast Fourier Transforms to detect workload patterns of the applications running in the IaaS layer and proposes a Markov model that predicts the incoming load. Their goal is to achieve "predictive migration" of the virtual machines, minimising energy usage without violating the SLA. Furthermore, [82] propose an architectural change on the IaaS layer by incorporating a "forecast engine", that is based on Seasonal Autoregressive Integrated Moving Average (Seasonal ARIMA) model.

This non-trivial model is based on assumptions such as the willingness of cloud users to "feed" it with the necessary data. The authors' claim is that, despite the difficulty in developing it, it can prove to be useful in deriving the resource requirements for each cloud user, resulting in optimal allocation of resources.

Moreover, [77] propose an architectural change on the IaaS layer by taking a control-theoretic approach. Their proposal consists of the use of three controllers (for the application, the memory and the CPU) which are integrated and run in parallel within a larger component (the authors name it the QoS controller), with the goal of optimising the performance of the applications that are hosted on the VMs. Other work by [80] proposes a "joint-VM provisioning" approach, where, by exploiting statistical multiplexing, multiple VMs are being provisioned by approximating their "combined" capacity needs. The common ground in the proposals described above is that they assume full control and observability of the IaaS layer, and are concerned only with fulfilling the objectives of the IaaS owner, which is not the case for a cloud application provider (e.g. SaaS provider) who might have limited control over the IaaS layer and different objectives from the cloud provider. In addition, it can lead to the misleading assumption that dynamic provisioning of resources is a matter of concern only to the IaaS owner.

This limited attention that has been paid to dynamic resource provisioning (elastic provisioning) for SaaS providers had not been addressed until recently, when researchers started to propose dynamic resource provisioning methods specifically for SaaS providers [83, 84]. This gap has also been noticed by [85], where the authors note that most of the work in provisioning resources in the cloud is focused on IaaS providers, and by [83], where the authors confirm the limitations of existing research, and describe the current work with respect to dynamic provisioning for SaaS providers as "in its infancy". These two papers make the important distinction between a SaaS and an IaaS provider, and do not assume that the SaaS provider is necessarily an IaaS provider.

Although this distinction may sound obvious, a significant amount of work in the cloud does not make this clear, and only lately have there been papers and surveys that define the cloud actors precisely. The observation made by [85] and [83] is very important, since it further emphasises the fact that more research is needed in various areas of cloud computing, especially from the application provider's viewpoint.

This area is mentioned because our research is concerned with the verification of dynamic provisioning from the application layer's perspective (e.g. SaaS). We believe that, in order for the dynamic resource provisioning proposals to be reliable, for use in practice and to advance the state of the art, they should be verifiable. As has been described in the previous section, the problem of managing resources in the cloud efficiently is strongly related to the decision maker whether it is a human who specifies threshold-based values manually or according to a mathematical model. Furthermore, instead of a human operator, it can even be a highly automated system which has been programmed to drive the application in the correct states. The main objective of those decision makers is to take full advantage of the cloud's elastic infrastructure in order to guarantee the QoS promised to the end users, and keeping the operational costs as low as possible.

Dynamic provisioning of resources is based on the elasticity property of cloud computing. Elasticity is a property that spans over dependability, performance, reliability and cost requirements. These requirements are called *non-functional requirements* and can be viewed either as global constraints of a system, or objectives that should be maximised or minimised accordingly. Current research that has been mostly focused on the area of provisioning resources has neglected the *correctness* criteria of the elasticity proposals; in fact we make a similar observation as the one made by [86] in 2014, who noted that most of the proposals that are related to automatic resource provisioning such as [81] and [87], "tend not to be accompanied by correctness guarantees".

This is further reinforced by the fact that, despite all of the on demand resource provisioning proposals, most providers still do not provide QoS guarantees with respect to the non-functional requirements in their SLA. As [88, 89] note, none of the public cloud providers provide performance guarantees to their end customers. It should be noted that there are some objective reasons for this, such as sudden and significant changes, hardware failures, etc., which makes it a challenging task.

The existing work on evaluating non-functional requirements from an application provider's perspective is based on two schools of thought: testing-based and formal verification-based approaches. In the following sections, we review the related work conducted in these areas and report on their limitations. Most of the focus is put on reviewing the existing work that has been conducted in the area of verification as it is more closely related to our research.

3.2 Testing Non-Functional Requirements for Cloud Computing

Non-functional testing is the process of testing the non-functional requirements of a system (e.g. performance, scalability, security, etc.). A problem in this process is the assessment of the test results, because it is harder to define to what extent the non-functional properties have been tested, compared to functional testing, where a true/false result is produced by a test oracle against a test case. This problem is of course harder to deal with within the context of cloud where there are many unpredictable factors, possibly unknown to the cloud application provider, making the quantification of quality a challenging task.

More specifically, the heterogeneous and unpredictable environment in which cloud-based applications operate puts a significant burden on the non-functional testing process because testers and decision makers have to determine to what extent the deviation from the expected behaviour is acceptable. This brings to the surface the problem of classifying emergent behaviour as acceptable or unacceptable.

The notion of emergent behaviour has been well described by Mogul [90], where he foresees the need that will arise in the future for *"developing testing techniques for emergent misbehaviour"*. This idea has recently been put forward by Elbaum and Rosenblum [91], who emphasise the inefficiency of existing testing methodologies, such as coverage criteria, in "exposing or controlling the sources of uncertainty". Another drawback of non-functional testing that limits its reusability is the tight coupling between the test cases and the underlying software. In a traditional non cloud-based scenario, this might be acceptable; however, this is not the case for cloud-based applications which are highly dynamic either due to the continuous updates from their provider or from the variations in the workload patterns that are being exercised by the tenants. This dynamicity could cause the test cases to be out of date very soon. For example, it would be extremely hard and costly to add new test cases that exercise new behaviour and to perform regression testing every time an update is being made, leading to a test suite that does not represent the behaviour of the software at a given point in time.

In an attempt to address the aforementioned challenges, there have been testing-based proposals in the literature which focused on applying existing non-functional testing methodologies for cloud-based applications, usually by adding them to the stack of the "as a Service" model [92, 93, 94, 95, 96, 97, 98]. This practically means that a portion of time-consuming and costly testing activities, such as re-running the test scripts for regression testing, is outsourced to a service provider who extrapolates the advantages of the cloud infrastructure. While the above proposals might seem interesting, they seem to neglect the dynamic along with the evolving nature of modern cloud-based applications that makes it virtually impossible to perform exhaustive testing, and in the case where this could happen the cost would have been prohibitive. Moreover, these proposals do not propose concrete solutions regarding the evaluation of non-functional requirements; rather their scope is general and they talk about the difficulties that manifest themselves in cloud-based non-functional testing from a general perspective. In particular, these proposals take a slightly more passive approach with respect to the various properties that need to be evaluated, without taking into consideration the time and cost constraints that are a part of cloud-based software.

Other proposals, such as [99] present more concrete ideas and propose a framework that is a combination between event-driven and combinatorial testing in order to perform regression testing on SaaS. Their main contribution is a coverage metric that is used to prioritise the test cases in regression testing. Although their work is interesting, it presumes the existence of a large enough test suite, something which is not applicable to new SaaS systems. In addition the evaluation is rather limited since it is based on a single system, and as a result the generality of the method is not guaranteed.

The work of [100], which takes a broad perspective and talks about best practices for engineering SaaS systems in general, includes a section on testing where it is argued that SaaS engineers could benefit from the work that has been conducted in software product lines, and the notion of a *Test-Tree* is proposed as a way to exploit similarities between tenants and consequently to minimise testing time. Similar to [100] are the works by [101, 102] which propose models inspired by software product line engineering as a way to cope with the evolving nature of SaaS. However, these works only propose a model, without evaluating it or showing how it could work in practice. Other testing-based approaches for evaluating non-functional requirements in SLAs are from [103], [104] and [105], where the authors propose a coverage criterion that assists in deriving the test requirements from the SLA, however they do not talk about SaaS or cloud-based software. Similar to the authors above, [106] proposes a genetic algorithm in order to generate test inputs that cause SLA violations, but does not take into account the unique characteristics of a cloud environment.

Moreover, work on the evaluation of auto-scaling policies includes [107], which proposes a performance metric for evaluating auto-scaling policies, but it is not clear where their experiments were conducted and to what extent their proposed metric can be helpful in a realistic cloud setting.

The work of [108] proposes a performance evaluation framework for evaluating auto-scaling strategies (PEAS). The authors formulate the evaluation of the auto-scaling strategies as a chance constrained optimisation problem, which is then solved through scenario theory in order to give probabilistic guarantees on the QoS. Despite the fact that the authors use realistic workloads, the evaluation of their framework is performed on their simulator where they attempt to simulate the cloud.

Furthermore, [109] perform an experimental evaluation of auto-scaling policies for complex workflows, by comparing various auto-scaling techniques. However, this evaluation is conducted on their own OpenNebula private cloud, and not on a public cloud provider. The main difference between the previous three works and our work is that they are not based on formal verification techniques and/or that the evaluation of these works is performed on a simulator or on a private cloud.

A work which has evaluated auto-scaling mechanisms on two public cloud providers (Amazon EC2 and Flexiscale) is the work by [110]. The authors experimentally compare Flexiscale's auto-scale mechanisms against those provided by Amazon EC2. However, as the authors note, this cloud provider did not have any built-in support for auto-scaling at the time, rather this is developed by themselves. As a result, it is not clear how well this matched the auto-scaling mechanisms of major public cloud providers, and whether reliable conclusions can be drawn.

Closing this section, we note that there is no prior work in the area of evaluating auto-scaling policies which combines aspects of formal verification with validation based on real data. Specifically, there is no prior work in which the auto-scaling policies are evaluated using formal verification techniques and which builds probabilistic performance models for horizontal auto-scalers where the models are: i) constructed and validated based on real data gathered from applications deployed on public cloud providers and while the *real* auto-scaler was executing; ii) taking into account aspects (e.g. capacity changes) of the documentation of major public cloud providers (Amazon EC2 and Microsoft Azure).

In general, however, we note that the landscape in testing-based approaches with respect to evaluating non-functional requirements for SaaS or cloud-based software is rather foggy and more research is needed, an observation made by [99, 100, 29].

3.3 Verifying Non-Functional Requirements for Cloud Computing

Formal verification techniques such as model checking have been applied successfully to a wide range of systems, from a functional perspective [69]. The purpose of a model checker is to algorithmically check the states of a model (e.g. finite state machine), to see if a functional property, usually expressed in a temporal logic formula, holds or not. However, guaranteeing absolute correctness for systems that operate in heterogeneous and unpredictable environments, such as the cloud, is too optimistic. In addition, the increasing complexity of systems that operate in these types of environments has shifted the focus from qualitative to quantitative verification [111, 72]. Quantitative properties include the non-functional requirements, and in the context of cloud computing, non-functional requirements are at the center of attention, since their satisfaction determines the QoS offered to users. Therefore, being able to provide performance guarantees for systems which operate in these types of environments is crucial; formal verification techniques based on formal modelling and analysis of probabilistic models [112] can help towards accomplishing this task [113]. A formal model can also help the various cloud application providers (e.g. SaaS providers) and their end users to resolve the ambiguity that stems from the use of natural language in the specification of SLAs that exists nowadays [76].

Probabilistic model checking has been employed with great success in recent years to verify and analyse properties of systems that manifest uncertainty. Domains include automotive systems [114], security [115], biology [116] and software engineering [117]. The latter domain, software engineering, includes works of researchers who take a broader view on software engineering problems, usually considering software (and systems) operating in a self-adaptive context [118]. In general, in a self-adaptive context the need to incorporate quantitative verification has been advocated by [117], in which the authors describe the stages of the adaptation process which can benefit from it. Other research works which use probabilistic verification in a self-adaptive context include the works of [119] and [120]. The former employs PRISM-games, an extension of PRISM, in order to develop stochastic game models to reason about different algorithms for self-adaptation; the latter reasons about the human factors that affect the adaptation process.

In cloud computing formal verification approaches have not been applied widely, and towards addressing this gap, a paper by [29] attempts to pave the way for future research with respect to verification of cloud services. Although this paper does not talk explicitly about formal verification, it proposes the use of model-based approaches in order to check non-functional properties of cloud-based SaaS systems. Furthermore, works of [117, 121] address the need for further research in the area of cloud computing and propose as an area for further research the continuous verification of non-functional properties. Lately, there has been an increased interest by researchers in applying probabilistic model checking to cloud computing. The main reason is that, from whichever perspective cloud computing is examined (e.g. IaaS, PaaS, SaaS layer), there is an inherent degree of uncertainty, and there is a clear need for this uncertainty to be quantitatively analysed.

Over the last years, there has been an active interest from researchers in employing probabilistic model checking to deal with the resource provisioning problems in the cloud. For instance, Fujitsu researchers [113] address the problem of assess-ing the efficiency and reliability of live migration operations, which are an integral part of the virtualised technology used in the cloud. To conduct this performance analysis, they employ probabilistic model checking to verify the performance of live migrations in the IaaS layer. However, they assume that migration requests are distributed in a uniform way, which is not necessarily true in practice [122].

In addition, [8] propose a Markov decision process model developed with PRISM, among other models, in order to formally verify different types of auto-scaling policies, including rule-based ones. The difference between our work and theirs is that we focus exclusively on rule-based auto-scaling policies, by developing one dedicated model to simulate the dynamics of the auto-scaling process. As a result, we take a vertical in-depth approach in the auto-scaling process, by considering a significant number of parameters that occur in realistic cases.

What further differentiates our work from others is that we perform an extensive validation of our models on two major public cloud providers (Amazon EC2, Microsoft Azure). This means that we do not have, or assume, any type of control on the underlying auto-scaling mechanisms or on the VM provisioning methods and strategies employed by the cloud provider, and through our model, we try to infer the different outcomes that could happen.

In general, however, we note that the use of probabilistic model checking for pragmatic cloud use cases is still at its infancy; our research aims at bridging this gap.

3.4 Kalman Filters for Resource Provisioning

The use of Kalman filters has been proposed for the effective provision of resources in the cloud or in other utility-based computing settings by [9, 10, 11]. These proposals usually come from advocates of the autonomic computing paradigm in which the Kalman filter is combined with a control system (i.e. controller), in order to provide an effective way of automating the resource-allocation decisions. The integration of a Kalman filter with a controller stems from the fact that the Kalman filter can be used as a *predictor* for tracking noisy performance parameters, such as the CPU utilisation. These predicted values are then given as inputs to the controller, thus allowing for a proactive resource provisioning approach to be taken. Furthermore, if predictions about the future state of the system are not of interest, the Kalman filter can be used purely as an *estimator* for estimating the performance parameters of the "current" state of the system. This is a particularly advantageous/appropriate use case for the Kalman filter since it is very effective in filtering out the noise from the true signal (e.g. CPU utilisation data).

The first use of a Kalman filter in a resource provisioning context was by [9], in order to track the parameters of a system which comprised a web server, a disk and a Common Gateway Interface (CGI). However, it is worth noting that the performance model (in the Kalman filter's terminology the performance model is referred to as the system model) of the system was abstracted in the form of a queueing model, and many of the parameters were assumed to be constant over time. Later, this work was extended by considering time-varying parameters, instead of constant ones, of the underlying performance model [9].

Systematic studies on the topic of employing Kalman filters for resource provisioning purposes were conducted by [10, 11, 123], in which the authors propose a control-based resource allocation approach where the Kalman filter was integrated with feedback controllers. In their work, the Kalman filter is used to track noisy CPU utilisation data and the controller's task is to maintain the CPU allocation accordingly. It is worth noting that the performance model, the CPU utilisation model in this case, in which it is assumed the emitted measurements originate, is a one-dimensional random walk. Specifically, they propose a Single-Input, Single-Output (SISO) Kalman controller, a Multiple-Input, Multiple-Output (MIMO) Process Noise Covariance Controller (PNCC) and an Adaptive MIMO PNCC. The first is used to dynamically adjust the CPU allocation of individual virtual machines, while the second one is used to take into account the resource couplings which exist in multi-tier applications. Finally, the third controller is configured in a similar fashion to the second one; however, the external noise which affects the system is assumed to be nonstationary in this case.

Similar related work includes [5] which proposes a stochastic model predictive control approach to explore the trade-offs that an application provider faces when renting resources from a cloud provider. Overall the problem is framed as an optimal control one, where the application provider tries to meet its Service Level Objectives while minimising its costs at the same time. The performance model is a queueing model used as an approximation for modelling the performance parameters of a multi-tier cloud application. In this context, the Kalman filter constitutes a component of the overall framework and tries to estimate the parameters of the performance model (e.g. the size of the queue).

Finally, [124, 125] propose the use of Kalman filters as part of a wider service, for proactive auto-scaling decision-making in the cloud. In the first paper, the authors challenge the existing rule-based auto-scaling approaches offered by the cloud providers. They propose a new cloud service, under the name Dependable Compute Cloud, which can determine the correct auto-scaling decisions to be taken, in a proactive manner. In the second paper, the same authors employ their modelling approach to analyse the workload impact on the various scaling decisions. In both of those papers, a variant of the Kalman filter algorithm is used to infer unobservable system parameters, such as service times and service rates. The system in their case refers to a multi-tier cloud application. However, this system is modelled as a queueing network where homogeneity among the servers and perfect load balancing are assumed, assumptions which do not necessarily reflect the operation of the system in a realistic setting.

3.5 Performance Analysis of Kalman Filters

Even though, from a theoretical point of view, the Kalman filter may give the impression that it will produce an optimal estimate, this might not be the case when applied to realistic situations. The reason is that the theory behind the Kalman filter often makes assumptions which could not be met in practice, such as the assumption that there is precise knowledge of the system and noise matrices, or that the numbers can be represented with infinite precision [15]. Specifically, it is assumed that the representation of the model used in the Kalman filter algorithm corresponds exactly to the "truth" model [16]. As a result, the approximation of the actual process which is expressed by the state space equations might lead to a phenomenon known as *divergence* [13] where the calculated state estimate from the Kalman filter does not represent the actual state of the process accurately. In consequence, there is a high potential for the estimated error covariance not to be consistent with the actual error covariance [126], and as a result the accumulated estimation errors will hinder the effectiveness/accuracy of the (Kalman filtering) estimation procedure (will render the estimation procedure obsolete). As an aside, it should be pointed out that model approximations of the actual process are not necessarily the result of an erroneous analysis, but also they can arise in a deliberate manner as a way to restrict the computational requirements of the Kalman filter [45] (e.g. reducing the order of the model), and as a result it becomes quite significant to quantitatively analyse this "degree of suboptimality"[43]. The *precise* identification of the conditions which could result in divergence is not a straightforward task, mainly because the whole concept of divergence is regarded more or less as a qualitative concept [35], which encompasses a whole spectrum of errors.

The causes of divergence are mostly attributed to either modelling or numerical errors [23, 43]. This is because the process of developing the Kalman filter in a computer implicitly means to "transfer" the rich mathematical theory which surrounds the Kalman filter to the computer's digital representation. As a result, when the outcome of this process is erroneous it is either because the constraints imposed on us by the computer (e.g. finite precision arithmetic) and the "digital" behaviour of the Kalman filter does not match the "theoretical" behaviour or because errors have been introduced during the modelling phase [127]. In particular, divergence because of numerical instability of the numerical algorithms used and because of modelling errors started to become apparent once the Kalman filter was gaining popularity in a growing number of engineering applications [128]. In the next sections we describe what is actually meant by numerical instability and modelling errors in the context of the Kalman filter.

3.5.1 Divergence Due to Numerical Instability

The part of the filter that could hinder its numerical stability, and so cause it to produce erroneous results, is the propagation of the estimation-error covariance matrix P in the time and measurement updates [23, 14, 13]. This is because the computation of the Kalman gain depends upon the correct computation of P and round-off or computational errors could accumulate in its computation, causing the filter either to diverge or slow its convergence [14]. While, from a mathematical point of view, the estimation-error covariance matrix P should maintain certain properties such as its symmetry and positive semidefiniteness to be considered valid, subtle numerical problems can destroy those properties resulting in a covariance matrix which is theoretically impossible [54]. Out of the two update steps in which the filter operates, the covariance update in the correction step is considered to be the *"most troublesome"* [13]. In fact, the covariance update can be expressed with three different but algebraically equivalent forms, and all of them can result in numerical problems [23].

In order for P to be statistically valid it must be (symmetric) positive definite. Briefly, this means that all of its eigenvalues are positive real numbers. This is for two reasons. First, from a modelling perspective, if its eigenvalues were zero, this would translate to a filter which may completely trust its estimates and consequently would avoid taking into account the subsequent measurements, placing all of its “belief” in the system model [23]. Second, from a numerical stability perspective, it does not suffice for the eigenvalues of P to be greater than zero, because if they are in close proximity to zero, then round-off errors could cause them to become negative, rendering it totally invalid [35, 14, 129].

In fact, the three equivalent forms to express the covariance measurement update are susceptible to numerical errors [23] and cannot guarantee the numerical stability of P. For example, the covariance update $P_k^+ = (I - K_k H_k)P_k^-$ is generally not preferred because it is too sensitive to round-off errors [23], which means neither the symmetry nor the positive definiteness of P_k can be guaranteed. That is because this update takes the product of nonsymmetric and symmetric matrices, a form which has been characterised as undesirable [13].

Alternatively, changing the covariance measurement update equation to $P_k^+ = P_k^- - K_k S_k K_k^T$ could potentially pose a *“serious numerical problem”* [13], such as P_k losing positive definiteness. Finally, while *Joseph’s stabilised form* [130], given by $P_k^+ = (I - K_k H_k)P_k^-(I - K_k H_k)^T + K_k R_k K_k^T$, is considered to preserve the numerical robustness of P^+, it is not totally insensitive to numerical errors [23]. An additional disadvantage is the high computational complexity, which is $O(n^3)$ [14, 13], since the number of arithmetic operations such as additions and multiplications is considerably higher compared to the simpler form. One might assume that numerical problems especially related to round-off errors were a problem of the past when the word length of processors was shorter. Of course, the advancements in technology such as the increased word lengths of processors have greatly ameliorated this situation and engineers today do not have to face to the same extent the numerical problems their predecessors had to face some decades ago. However, failures because of numerical problems are still a significant issue if the Kalman filter is implemented in embedded systems or any other computing device with stringent computational requirements [15]. Moreover, it has also been shown that numerical problems can still exist even in today's Kalman filter implementations because of round-off errors [14, 15, 17]. In order to demonstrate the numerical problems which could arise, consider the following example, which is an extension of an example from [14].

Example 3.1. Let us first define a numerical quantity δ, which is the so-called *machine epsilon* (*eps*) or *unit round-off error* and is defined as the distance between 1.0 and the next largest double precision number. For instance, for a double-precision number this number will be $2.2204e-16$, or equivalently 2^{-52} in the binary numeral system which means that a number can be stored with a precision of roughly 16 digits. Alternatively, it can be thought of as an upper bound on the relative error between a number and its computer-based representation. It can be proven that for the cases where rounding is implemented that $\frac{|\Delta x|}{|x|} \leq \frac{\delta}{2}$ [131], where Δx is the true error between a number and its machine approximation.

Consider the innovation covariance update in the measurement update part of the Kalman filter (for more details see Section 2.4), where the measurement $R = \delta^2 I_2$, $\delta > eps$ and $\delta^2 < eps$.

$$P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad H = \begin{bmatrix} 1 & 1 \\ 1 & 1+\delta \end{bmatrix} \tag{3.1}$$

For our example, we have set the value of $\delta = 10^{-8}$, since it satisfies the constraint which was set beforehand ($\delta > eps$ and $\delta^2 < eps$).

Performing now the computations for the innovation covariance update given by $S = R + HP^{-}H^{T}$ will yield the following matrix:

$$S = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \tag{3.2}$$

We note that the matrix S is singular, which means that it cannot be used in the subsequent Kalman gain equation because it is non-invertible. The case explained in the example above is not the only one which can be affected by numerical errors. In fact, in our work we totally avoid numerical errors which can arise from the inversion of S by processing the measurements in a sequential manner which allows for the division operation to be used in the place of an inversion. By doing that, we restrict the range of the different possibilities which can cause numerical errors and focus our efforts on the analysis of the estimation-error covariance matrix P.

3.5.2 Divergence Due to Modelling Errors

Divergence detection falls under the wider scope of estimation theory, and more specifically to what is often referred to as *consistency evaluation* of estimators. An estimator is said to be *consistent* when its estimates will converge to the true value as the sample size increases [23], or informally when its estimates become increasingly better over time. Also, it is important to consider the context under which the Kalman filter operates. For example, the *consistency tests* that can be applied to a Kalman filter operating in a simulation-based environment cannot be applied to the situation where the Kalman filter operates in *real-time*, and processes *real-time data* [23]. This is because devising statistical tests based on the *actual* estimation errors to assess the consistency of a Kalman filter is only applicable in simulations, where the true state of the system is known. A popular way to deal with divergence problems, once they are detected, is called *filter tuning.* This works by adjusting the process noise covariance matrix Q, which the Kalman filter uses to represent the unknown effects affecting the system whose state is to be estimated. This can be thought of as injecting “pseudo-noise” and is considered one of the hardest and most challenging tasks in developing an operational filter [132, 133, 42], because it encompasses all the unknown effects that perturb the process model of interest. It can also be thought of as a modelling error compensation technique [134, 23, 43, 135, 45, 35], and can be applied to a self-adaptive context, where it is sometimes referred to as *self-tuning*

[13]. This context is often referred to as adaptive filtering which means that the Kalman filter adjusts its noise levels dynamically according to some performance criterion. Furthermore, filter tuning can also happen in the design phase, e.g. using a simulation-based environment and an iterative search to select parameter values that will achieve the best possible estimation result [13].

In this work, we instead focus on a quantitative analysis of the *innovation* (or *measurement residual*), which represents the difference between the actual and the predicted system measurement. This is considered a "*prime indicator*" of filter divergence [35]. The *consistency criterion* of the innovation sequence we consider is that its computed covariance from the Kalman filter (e.g. actual statistics) is consistent with the design statistics [35]. In other words, we want to verify that the actual statistical properties are consistent with the theoretical statistical properties.

To the best of our knowledge, there is no prior work applying probabilistic model checking to Kalman filters. Perhaps the closest is the use of non-probabilistic model checking on a variant of the Kalman filter algorithm in [136], which applied model checking to target estimation algorithms in the context of antimissile interception. More generally, others have applied formal methods to state estimation programs. For example, [137, 138] combined program synthesis with property verification in order to automate the generation of Kalman filter code based on a given specification, along with proofs about specific properties in the code. Other work relevant to the above includes [139], which used the language ACL2 to verify the loop invariant of a specific instance of the Kalman filter algorithm.

CHAPTER 4

Quantitative Verification of Cloud-based Auto-Scaling Policies

In this chapter, we present a novel approach based on quantitative verification to produce performance guarantees on particular rule-based auto-scaling policies for cloud systems. In particular, we use probabilistic model checking, which is a formal approach to generating guarantees about quantitative aspects of systems exhibiting probabilistic behaviour, and the tool PRISM. Guarantees are expressed in quantitative extensions of temporal logic and construction, and numerical solution of probabilistic models is used to quantify these measures precisely.

We demonstrate the usefulness and efficiency of our techniques through a detailed validation process on two public cloud providers, Amazon EC2 and Microsoft Azure, targeting two cloud computing models, Infrastructure as a Service (IaaS) and Platform as a Service (PaaS), respectively. Another important novel aspect of our approach is the combination of probabilistic model checking with *Receiver Operating Characteristic (ROC)* analysis during empirical validation. This allows us not only to refine our original probabilistic estimates after collating real data and to validate the accuracy of our model, but also to obtain global QoS violation thresholds for the policies.

Our modelling and verification framework is intended to minimise the time and costs for cloud application owners who do not have the resources or desire to clone their existing applications in order to test them in a cloud-based environment. It can also provide valuable assistance in designing, analysing and verifying the auto-scaling policies of applications and services deployed on public clouds, and could help in providing formal performance guarantees.

The remainder of this chapter is structured as follows: In Section 4.1 we present an overview of the modelling and verification framework. Next, in Section 4.2 we describe our formal modelling approach to constructing a probabilistic model of the auto-scaling process. Then, in Sections 4.3 and 4.4 we present our techniques for applying probabilistic verification to the model and validating the results, respectively. In Section 4.5 we reflect on the results and in Section 4.6 we summarise the main contributions of this chapter.

4.1 Overview of the Framework

In this section we give an overview of the modelling and verification framework for building probabilistic performance models for horizontal auto-scalers on Amazon EC2 and Microsoft Azure, which are then used for the verification of auto-scaling policies. First, we present the workflow of our approach as a sequence of steps which have to be carried out in order to apply the framework. A high-level overview of this process is also illustrated in Figure 4.1. Then, we state the assumptions under which the framework has been developed, and we conclude by describing the conditions under which a user of the approach can directly use the specific DTMC models described in this chapter.

Overall approach and workflow. The first step potential users of our approach should perform in order to apply it to their particular problem domain is some form

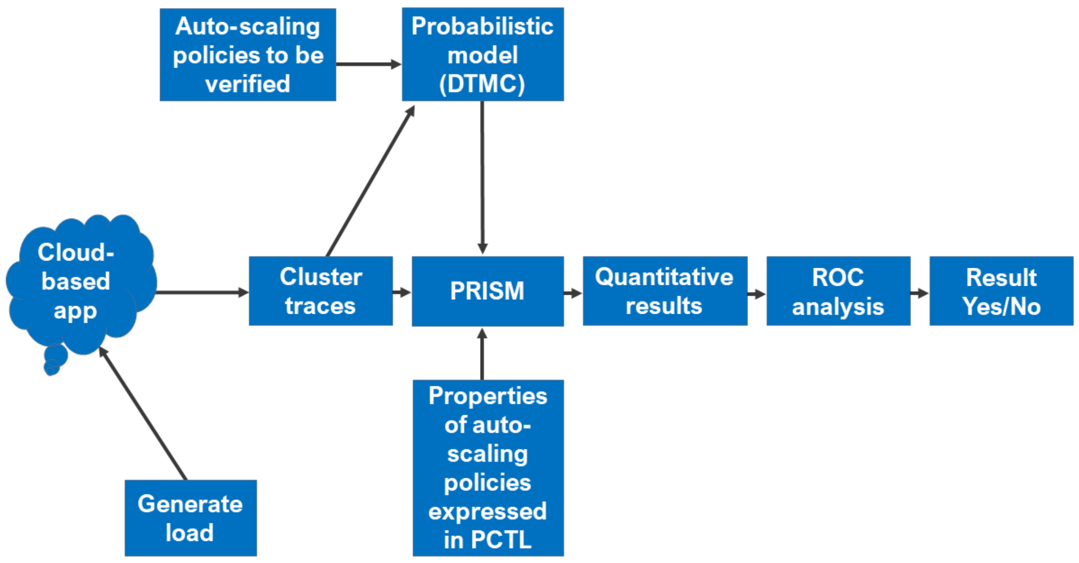

Auto-scaling policies to be verified
Probabilistic model (DTMC)
Cloud-based app
Cluster traces
PRISM
Quantitative results
ROC analysis
Result Yes/No
Properties of auto-scaling policies expressed in PCTL
Generate load

Figure 4.1: A high-level diagram of the proposed framework.

of load identification, in terms of the type of workload that will affect their cloud-based application. This is particularly important since the states in the probabilistic model will store information about the CPU utilisation and response time for a number of VMs under a particular workload. In our case, for example, the minimum number of VMs we consider is 1 and the maximum is 8 for the Amazon EC2 case, while the minimum and maximum number of VMs considered for the Microsoft Azure case is 1 and 4, respectively.

The next step is to generate load to the cloud-based application to gather the necessary CPU utilisation and response time traces. For this step, in our work we use the Apache JMeter [140], which allows us to simulate the type of load we are interested in modelling, on the VMs. Furthermore, the load patterns used in our case are a periodic and an "aggressive" load pattern. More information about these load profiles is given in Sections 4.4.3 and 4.4.6, for the Amazon EC2 and Microsoft Azure cases, respectively. Once CPU utilisation and response time traces have been obtained, the clustering process begins which effectively determines the number of states (per VM number) in the probabilistic model.

As soon as the number of states is determined, the next part is the development of the DTMC in PRISM. The auto-scaling policies to be verified should be used as a guide for this step in order to allow for an accurate representation of the transitions between the states in the DTMC.

Following the construction of the model, the performance properties, expressed in PCTL, are verified against the model which abstracts the auto-scaling process, and the quantitative results are obtained. The more expressive the model is the more verification scenarios can be analysed. Next, ROC analysis is proposed as a way not only of validating the results of the model against new data obtained from the cloud-based application, but also of discriminating between auto-scaling policies which could result in a QoS violation (Result=Yes) or no violation (Result=No).

Assumptions and constraints. Our framework is targeted at cloud-based applications which operate web or application servers (hosted as VMs) to serve static or dynamic content to their users, respectively. As a result, we do not focus on specific software implementations of the above architecture. An assumption we make for the potential users of our framework is that they must have some prior knowledge about the total number of VMs that their cloud-based application requires. Moreover, out of the two types of auto-scaling (horizontal and vertical) which have been discussed in Section 2.1, our focus is on constructing probabilistic performance models for horizontal auto-scalers for Amazon EC2 and Microsoft Azure, and as such the auto-scaling policies we encode and verify in PRISM are horizontal. In terms of horizontal auto-scaling, our framework can be used to model and verify auto-scaling policies of various step adjustments and VM types. However, we do assume that the performance metric, on the basis of which the auto-scaling action is taken, is CPU utilisation.

Furthermore, the auto-scaling policies are verified against properties which relate to CPU utilisation and to response time. This means that the framework operates under the assumption that the user is interested in optimising for availability and not, for example, for cost. In our work we aim to strike a balance between availability and cost by considering an upper bound on the number of VMs as shown in Sections 4.3.1 and 4.3.2.

Both models are available online at [20]. For a user modelling a scenario similar to ours, parts of our models can be changed quite easily in order to use them directly. For example, the user would need to change the values of the minimum and maximum number of VMs, assuming that the lowest number of VMs is not 1. If we assume that the maximum number of VMs changes to 9 for the Amazon EC2 case, then the user would need to copy the existing PRISM code for a lower number of VMs (e.g. 8) and add it after the code of the eighth VM.

Also, if the step adjustments of the auto-scaling policies change, then the lower and upper bounds of CPU utilisation, on the basis of which the auto-scaling policy is taken, would need to change as well. We also assume that the models do not rely on a particular workload since the users can assign their own values of CPU utilisation and response time, as well as the probability values, based on their own data gathered for their particular workloads. However, under different conditions (e.g. another cloud provider) a different model would need to be built. This is because the models published online use computations with respect to how the changes in VM capacity occur or to how “flapping” situations are prevented, based on the Amazon EC2 [1] and Microsoft Azure [141] documentation, respectively.

4.2 Formal Modelling of Auto-Scaling Policies

The states in our models represent the information needed to capture the dynamics of the auto-scaling process and its impact on QoS. Apart from the use of boolean variables which are used to synchronise certain transitions in our models, we have employed clustering methods to summarise the monitored CPU utilisation and response time traces. These clusters are used to characterise the different states of our models, and in the following subsections we provide the details with respect to the clustering procedures followed.

We model the dynamics of the auto-scaling process as a discrete-time Markov chain, the states of which are updated in each time step, corresponding to one minute and five minutes for the Amazon EC2 and Microsoft Azure cases, respectively. For both of these cases, we have discretised time into slots of greater than or equal to one minute, since there has to be a sufficient duration to gather and analyse the "external" data that constitute the state (e.g. arrivals, utilisation levels). In addition, due to the general uncertainty that manifests in a cloud environment, time intervals of less than one minute would make limited sense. They would result in unmanageably large spaces due to, for example, small temporal spikes in utilisation levels which can be ignored in our models. It would also be unrealistic to assume that a time step of less than one minute is long enough to flag a violation, and consequently to send an auto-scale request.

The states in our models can be divided into *deterministic* and *stochastic*. The former model the deterministic aspects of an auto-scaling policy that is to be verified. This is similar to a real case, where the cloud application will transition to states that have been previously defined by the application manager in the auto-scaling policy. To make the analogy more concrete with a realistic example, deterministic states in our model encode the subset of the conditions that apply to a cloud-based application, which determine whether an auto-scale request will be sent to the cloud provider or not. Stochastic states encode the probabilistic outcomes or responses of the auto-scale requests. The models have been developed in the PRISM modelling language (as opposed to the `ModelGenerator` API) and have been made available online [20].

4.2.1 Clustering of CPU Utilisation and Response Time Traces

For the model developed for the IaaS use case on Amazon EC2, we standardise the CPU utilisation and response time values by computing their *z-scores* [142]. Then, the model is initialised after *k-means* clustering has been run on the CPU utilisation and response time traces. The value of k is also the number of different outcomes that could happen when a scale-out or a scale-in action occurs. Equivalently, k can be thought of as the number of states per the number of VMs in operation. In a sense, it captures the CPU utilisation and response time variability that exists for a given number of VMs in operation.

As a result, as the size of k grows larger, the more detailed the possible state representations will be, possibly at the cost of adding a degree of overhead in the verification process. Conversely, for smaller values of k, the scalability of the verification process is improved, at the possible cost of representing the states in a "coarser" way.

In our model of the IaaS use case on Amazon EC2, after experimenting with different cluster sizes, we have set $k = 5$. For the second type of model which targeted a PaaS use case on Microsoft Azure, a *univariate* clustering method was employed, since we did not take into consideration the response time, and consequently had to deal with one-dimensional data. This allowed us to use the *Ckmeans.1d.dp* algorithm which "guarantees optimality" for data in a 1-D space [143]. The value of k in this case was determined by trying a range of different cluster sizes and selecting one with the appropriate *Bayesian Information Criterion (BIC)* (see Figures 4.4 and 4.5 for the BIC and the respective clustering plots for the CPU utilisation traces for the auto-scaling policies with the 5-minute cool-down period considered).

4.2.2 Encoding Auto-Scaling Policies in PRISM

Although two different models have been developed for the Amazon EC2 and Microsoft Azure case, the core principles underlying them are essentially the same. For example, both of the models expect an auto-scaling policy as an input, the creation of their states relies on clustering, and the "philosophy" of how transitions unfold between the states is similar. In addition, both of the models share the same high level objectives, since they are meant to assist in the auto-scaling decision making process. In the next paragraph, we describe the important building blocks of our models by highlighting their differences and by explaining the reasons for those differences.

For the Amazon EC2 case, as shown in Table 4.1, the "free" parameters (*constants*), which are left to the user of the model to set, are: i-iv) step adjustments for scale-out and scale-in rules; v) the increment specifying the number of instances which will be added; vi) the decrement specifying the number of instances which will be removed; vii) the number of VMs that are currently reserved; viii) the maximum time (*MAX_TIME*) the model will run in terms of t; ix) the probability p of delay, defined as the number of time steps t set in the *WAIT_TIME* variable, in serving an auto-scale request; and x) the time that it will take for an auto-scale request to be satisfied. The first seven constants are under the control of the application provider and represent the values that an application owner would have to set in reality. The last two represent parameters that are not controllable and are being used as a basis for modelling and analysis of scenarios of interest (e.g. worst-case scenarios). All the model parameters for the Amazon EC2 case are shown in Table 4.1.

The "free" parameters for the model developed for the Microsoft Azure case are the increment and decrement variables, the number of currently reserved VMs and the maximum time the model will run. The reason for having fewer constants than

Table 4.1: Model Parameters for Amazon EC2

Model Parameters		
Time variables	*t* *WAIT_TIME* *MAX_TIME*	Discretised time (step) for each state Wait. time Max. time
Virtual machines	*INITIAL_VMs* *cVMs*	Number of reserved VMs VMs operating currently
Auto-scale reqs.	*scale_out_1* *scale_out_2* *scale_in_1* *scale_in_2*	Scale-out req. for CPU util. between 60% and 70% Scale-out req. for CPU util. between 70% and 100% Scale-in req. for CPU util. between 30% and 40% Scale-in req. for CPU util. between 0% and 30%
Other actions	*wait*	No action is taken
Auto-scale resp.	*sat_s_out_req* *sat_s_in_req*	Adds the capacity (%) requested Removes the capacity (%) requested
Adjustment	*s_o_adj_1* *s_o_adj_2* *s_i_adj_1* *s_i_adj_2*	Capacity(%) to be added when *scale_out_1* is chosen Capacity(%) to be added when *scale_out_2* is chosen Capacity(%) to be removed when *scale_in_1* is chosen Capacity(%) to be removed when *scale_in_2* is chosen
Boolean variables	*scale_out_trigger* *scale_in_trigger* *capacity_added* *capacity_removed*	Coordinates scale-out requests Coordinates scale-in requests Indicates that capacity has (not) been added Indicates that capacity has (not) been removed
Perf. metrics	*arrivals* *served_reqs* *util* *r_t*	Arrivals (requests) Average requests/jobs served per 1 time unit Average VM CPU utilisation Average response time
Probabilities	*q* *p*	Prob. of *util* and *r_t* (based on k-means) Probability of delay in serving an auto-scale request
Incr. / Decr.	*inc* *dec*	Scale-out policy adds instances in increments of *inc* Scale-in policy removes instances in decrements of *dec*

the Amazon EC2 model is to explore the impact on the performance metrics of interest of more "direct" auto-scaling policies; that is, policies which do not express the capacity to be added/removed as a percentage of the total capacity, but as an integer which specifies the number of VMs to be added or removed. In addition, the capacity adjustments are expressed as a percentage, as shown in the "Adjustment" columns of Table 4.2, where 0, ± 10 and ± 30 are the percentages of the total capacity to be added or removed accordingly. In this case, one has to think about the rounding rules that exist, and the conditions that have to be fulfilled when a percentage in capacity has to be added or removed.

In order to be able to develop a representative model for the Amazon EC2

Table 4.2: An example of a step adjustment auto-scaling policy as seen in Amazon EC2's documentation [1]

Scale-out policy			
Lower Bound	Upper Bound	Adjustment	Metric value
0	10	0	$50 \leq value < 60$
10	20	10	$60 \leq value < 70$
20	null	30	$70 \leq value < +\infty$

Scale-in policy			
Lower Bound	Upper Bound	Adjustment	Metric value
-10	0	0	$40 < value \leq 50$
-20	-10	-10	$30 < value \leq 40$
null	-20	-30	$-\infty < value \leq 30$

case, there are certain conditions with respect to the addition/removal of VMs that have to match those in real clouds. As an example we show some of these conditions, encoded as formulae in PRISM and invoked in the respective states in our model. Specifically, some of the conditions our model has to fulfil with respect to the scale-out and scale-in adjustments are the following. Values in the intervals [0..1] and ($-\infty$..-1) are rounded up, while values in the intervals [-1..0] and (1..$+\infty$) are rounded down, respectively. In Listing 4.1, we show a sample of the PRISM code for these conditions, expressed as formulae, as these are defined in Amazon EC2's documentation [1].

```
formula adjust_c1=(cVMs*s_o_adj_1)>=incr?ceil(cVMs*s_o_adj_1):incr;
formula adjust_f2=(cVMs*s_o_adj_2)>=incr?floor(cVMs*s_o_adj_2):incr;
formula adjust_si_f1=(cVMs*s_in_adj_1)>=decr?ceil(cVMs*s_in_adj_1):decr;
formula adjust_si_f2=(cVMs*s_in_adj_2)>=decr?floor(cVMs*s_in_adj_2):decr;
```

Listing 4.1: Changing capacity formulae

As a result, part of our reasoning when developing the model for the Microsoft Azure use case was to explore how much of the computational overhead of checking the changing capacity formulae for the scale-out and scale-in adjustments could be avoided. This is important when dealing with auto-scaling policies where the requested capacity is expressed as an integer, denoting the number of VMs to be added/removed.

After these parameters have been specified, the model transitions with a probability q to the possible outcomes of CPU utilisation and response time which are associated with the particular number of VMs. For the Amazon EC2 model, these outcomes, and the respective probability q, are set according to *k-means* clustering which has been run beforehand.

Then, depending on the granularity of the auto-scaling policy, there can be multiple states each of which is a deterministic state. For example, for the step adjustment auto-scaling policy in Table 4.2, the deterministic state would have had six different representations (one for each CPU utilisation interval). Then, depending on the utilisation level (similar to a real auto-scaling policy), the model transitions to the appropriate deterministic state, which causes the respective auto-scale request to be triggered or not, resulting in a transition to a stochastic state. This represents the fact that the auto-scaling request has been sent to the cloud provider, and is ready to be processed.

According to the type of scenario one wishes to analyse and verify, different transitions will occur. For example, let us assume that the probability p is set to 0.2, which means that with probability 0.2 the auto-scale request will be satisfied in the next time step, and with probability 0.8 the auto-scale request will be satisfied according to the $WAIT\_TIME$ which is associated with p. Then, for the first case the model will transition with probability p to a state where the *capacity_added* variable will be set to true, and subsequently the auto-scaling request will be immediately satisfied, resulting in a change in the overall VM capacity.

On the contrary, with probability $1 - p$, the model will transition to a state where the auto-scale request will be satisfied based on the best-effort reservation policy, and the respective boolean auto-scaling variables will be set to true/false accordingly, representing the realistic case in which the auto-scaling actions are being locked, and the application is forced to wait until the auto-scale request is satisfied.

For example, when a *scale_out* transition happens, the corresponding state variable *scale_out_trigger* is set to *true*, preventing further scale-out or scale-in requests from being triggered, as would have happened in a realistic setting.

In Listing 4.2 we show a sample of the PRISM code for the transition based on the value of p, which takes into account the stochasticity of auto-scale requests, by transitioning between the different waiting times with a probability p, and can be specified by the user a priori, or left as a free parameter in order to probabilistically verify an auto-scaling policy under different values of p. The $\rightarrow$ symbol is called a "guard", and the lines of code preceding it show the predicates that have to be satisfied for the transition to occur. In Listing 4.2, we show how this case is expressed in PRISM's *guarded commands* [144].

```
[choice](...)&(t<MAX_TIME)&(scale_out_trigger=true|..)&(best_effort=false)
&(imm_res=false)−> p:(imm_res'=true) + 1−p:(best_effort'=true);
```

Listing 4.2: Transition based on probability p

Also, the waiting time which was initialised before the model started will now start to decrement, and until the duration of the waiting time has elapsed, the model will be prohibited from sending any other auto-scale requests. Again, this is according to realistic cloud controllers in practice, where the application owner is prohibited from sending any more auto-scale requests until the one that has been sent has been satisfied. In Listing 4.3, PRISM sample code for the *scale_out_1* transition is shown.

```
[scale_out_1] (...) & (cVMs>=1)&(scale_out_trigger=false)&(scale_in_trigger=false)
&(best_effort=false)&(t<MAX_TIME)&(util>=60 & util<70)−>
(scale_out_trigger'=true)&(t'=t+1)&(actual_util'=ceil(util));
```

Listing 4.3: Scale-out step 1 transition in PRISM

Furthermore, while the waiting time is active, the model continues to generate load for the VMs to process. As has been described above, we abstract the generated load in the model, and our focus is directly on the impact of the load on the performance metrics (CPU utilisation and response time), based on *k-means*. The transitions unfold in a similar manner; for example, when the waiting time has completely elapsed, the auto-scale request will be satisfied, and the requested capacity will be added or removed.

For the Microsoft Azure model, there is no probability associated with the time it takes to satisfy an auto-scale request. This is because we did not want to investigate *what-if* scenarios with respect to the time variability of the auto-scaling policies, rather our focus was on analysing the impact on the QoS of varying the cool-down periods of an auto-scaling policy. Based on this intention, we added specific guarded commands to states in our model where a scale-in action is to be taken, to prevent "flapping situations" where the auto-scale controller triggers actions continuously [141]. For example, in the states where a scale-in action is to be taken, we evaluate if the resulting CPU utilisation after releasing a VM will be less than the scale-out CPU utilisation threshold, to determine whether the scale-in action will be skipped or not. In Listing 4.4, we show a code fragment of how this case is expressed in PRISM's guarded commands, and as a result the respective scale-in transition is skipped when the guard is violated.

```
[scale_in_act1] (...)&
(INITIAL_VMs=2)&
(cVMs=3 | cVMs=4)&
(decr=1)&
((util∗ cVMs) / (cVMs−decr) < cpu_threshold)
```

Listing 4.4: A fragment of a guarded command in scale-in states to avoid "flapping".

4.3 Formal Verification of Auto-Scaling Policies

In this section we present the verification process for the two cases considered, namely the IaaS case on Amazon EC2 and PaaS case on Microsoft Azure.

4.3.1 IaaS Case on Amazon EC2

The verification process consists of two phases. In the first phase, we generate load on the rented VMs to gather at least 100 data points for CPU utilisation and response time, for each VM number between 1 and 8. These 100 data points correspond to approximately 100 minutes of load generation, monitoring, and data gathering for each VM, and for each load type, resulting in approximately 26 hours of data collection. These data points are used for the initialisation of our model. In the second phase, we use *k-means* clustering to cluster the respective data points, which correspond to different outcomes of CPU utilisation and response time. Once the clustering process is finished, the clusters are fed into our model in PRISM, and once an auto-scaling policy or a set of auto-scaling policies is given as input to our model, we obtain the verification results.

Throughout this process we obtain CPU utilisation and response time guarantees under two load patterns: periodic and "aggressive" (see Section 4.4.3 for details). Specifically, we are interested in computing the probability that the cloud application (consisting of all the VMs) will end up in a state where a QoS violation exists. This is defined for states where the CPU utilisation is $\geq$ 95%, or the response time is $\geq$ 2 seconds for periodic load, or $\geq$ 5 seconds for the "aggressive" load (Listing 4.5) under the policies shown in Table 4.3. We vary the *INITIAL_VMs* in the range [1..8] and *inc*, *dec* in the range [1..3].

Table 4.3: Auto-scaling policies for formal verification.

Action	Inc/Dec	Min Util.	Max Util.	Initial VMs	Adjust.
Scale-out	[1..2]	60%	70%	2,3,4,8	[+10%]
Scale-out	[1..2]	70%	100%	2,3,4,8	[+30%]
Scale-in	[1]	0%	30%	2,3,4,8	[-30%]
Scale-in	[1]	30%	40%	2,3,4,8	[-10%]
Wait	-	40%	60%	2,3,4,8	0%

In Listing 4.5 we show how these properties are expressed in PRISM's syntax.

```
P=? [F util >= 95] //both load patterns
P=? [F r_t >= 2] // periodic load
P=? [F r_t >= 5] // "aggressive" load
```

Listing 4.5: Properties to be checked

In addition, since the outcome of the auto-scaling action depends on uncontrollable parameters, we vary the probability that the auto-scale request will be served in the next time step, p, and the *WAIT_TIME* in our model. It is important to note that the verification process is not driven by too "optimistic" or too "pessimistic" parameter tuning. Specifically, we are interested in verifying for which set of "reasonable" variables in the auto-scaling policy, utilisation and response time guarantees hold. By "reasonable", we mean that we do not analyse auto-scaling policies under unrealistic conditions, such as choosing an increment of 20 VMs, since there is a non-negligible cost associated with renting the VMs. Also, we avoid unrealistic assumptions with respect to the time taken to satisfy auto-scale requests, by not varying p above 0.5. This fits our purpose of performing worst-case analysis as well. In our modelling and verification approach the *worst-case* is defined for the situation in which the auto-scale request will never be satisfied immediately ($p = 0$), and it would take at least 5 (*WAIT_TIME*$= 5$) time units to be satisfied. Conversely, the *best-case* is defined with $p = 0.5$ and *WAIT_TIME*$= 1$. In Figures 4.2 and 4.3, we show the output of verification, for a specific auto-scaling policy, under periodic load. For the controllable parameters, we set the adjustments to $\pm 10\%$

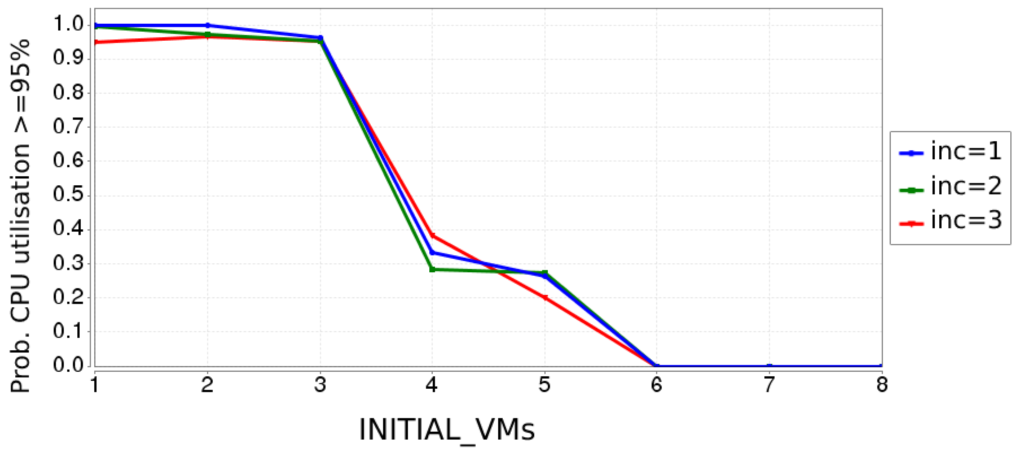


Figure 4.2: PRISM results for P=? [F util $\geq$ 95] (periodic load).

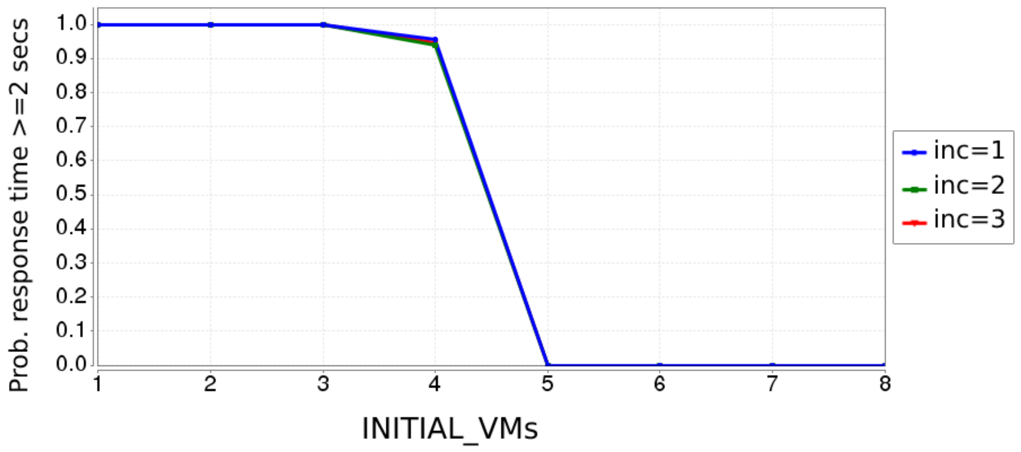


Figure 4.3: PRISM results for P=? [F r_t $\geq$ 2] (periodic load).

and $\pm$30% for the two steps, respectively, as in Table 4.3. For the uncontrollable parameters, we set $p = 0.2$ and *WAIT_TIME*= 3, as an average case, which is not biased towards worst- or best-case scenarios. In Figure 4.2, we can observe that the probability for a CPU utilisation violation follows a decreasing trend as the number of *INITIAL_VMs* increases. An important observation which we capture is the decrease by approximately 0.5 in the probability for a CPU utilisation violation when the cloud application starts serving HTTP requests with 4 VMs compared to 3. In Figure 4.3, the probability for response time violation fluctuates around 0.95, and drops sharply to 0 when the number of *INITIAL_VMs* becomes 5.

4.3.2 PaaS Case on Microsoft Azure

Table 4.4: Auto-scaling policies for formal verification.

Action	Inc/Dec	Min Util.	Max Util.	VMs	Cool-down
Scale-out	[1]	71%	100%	[1..4]	5 minutes
Scale-in	[1]	0%	39%	[1..4]	5 minutes
Scale-out	[1]	71%	100%	[1..4]	1 minute
Scale-in	[1]	0%	39%	[1..4]	1 minute
Wait	-	40%	70%	[1..4]	-

The verification process for Microsoft Azure follows a similar approach to the one for Amazon EC2, with the difference being that we did not consider response time as part of our QoS metrics, since we focused exclusively on CPU utilisation and how it is affected by the variation of the cool-down period, which is part of an auto-scaling policy. The reason for focusing exclusively on CPU utilisation is the fact that we wanted to restrict our attention to server side metrics, and only to metrics on which the auto-scale decisions are taken. The model was initialised in a similar way to the Amazon EC2 case.

In the first phase, we generated load to the VMs and we gathered approximately 400 data points for CPU utilisation for the range of VMs 1 to 4, resulting in approximately 26 hours of CPU utilisation traces that were used to construct our model. For example, in Figure 4.4 we show the BIC plots that were used to determine the number of clusters per VM number, and in Figure 4.5 we show the respective (optimal) clustering plots of CPU utilisation after having chosen the number of clusters according to the previous procedure. Our verification goal is twofold; first we wish to compute the probability that the auto-scaling policies in Table 4.4 could result in a QoS violation, and then to identify the set of states which have a high probability of transitioning to a "bad" state. By doing that, we show how PRISM can be used to narrow down the search state space to states which are more likely to cause "bad" auto-scaling actions. We define a "bad" auto-scaling decision as an auto-scaling request which, even though it has been successful and the requested capacity has been

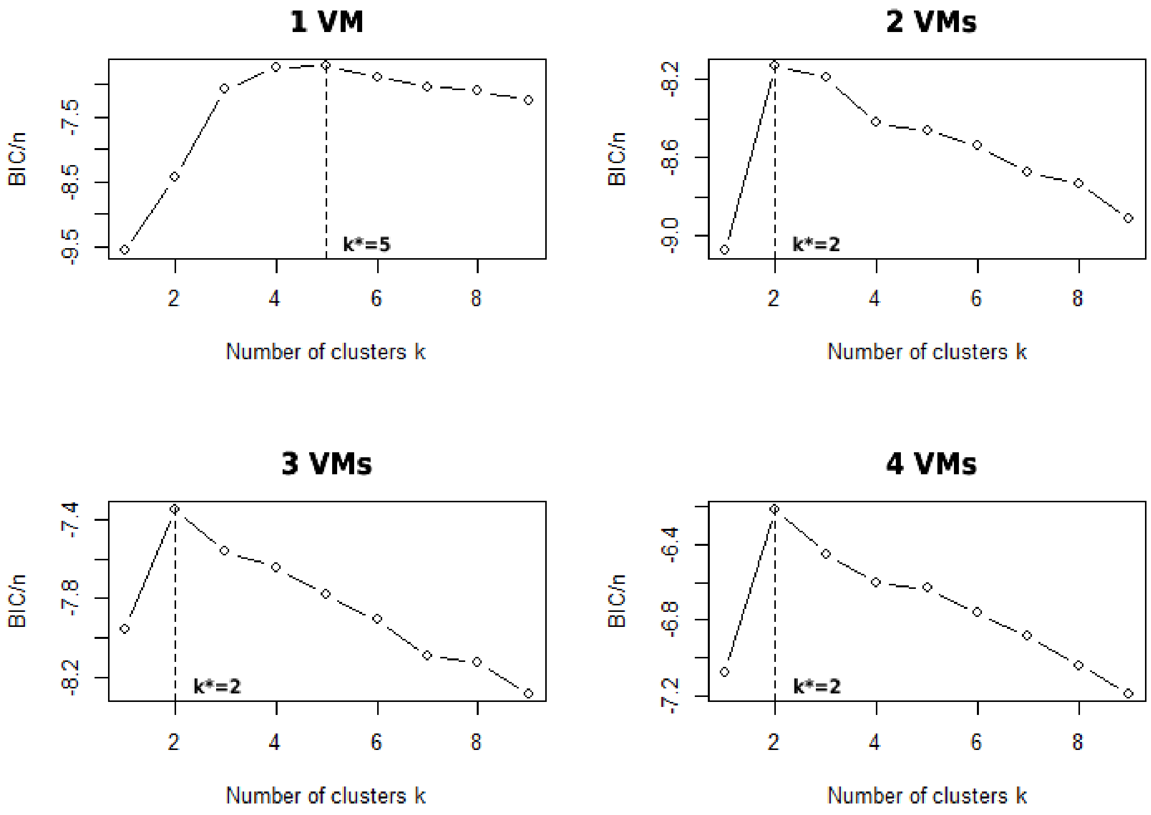


Figure 4.4: Determining k according to the Bayesian Information Criterion (BIC), normalised by sample size.

added/removed, still it has caused a QoS violation. Our main claim here is that properly configuring the temporal parameters is equally important to the proper configuration of the "standard" auto-scaling parameters such as the capacity to be added/removed.

To capture "bad" auto-scaling decisions, we first identify the set of states where a "bad" auto-scale state is true. This is achieved by adding two boolean variables in our model; *bad_scale_in*, and *bad_scale_out* and setting them to true and false when certain transitions occur in our model. After this point we can follow two possible verification strategies. We can either use PRISM's *reward* operator `R` to assign a reward of 1 to states labelled as "bad" and then compute the expected value of "bad" auto-scale decisions over all paths for a given period. An alternative verification strategy is, after having computed the probabilities for the "bad" auto-

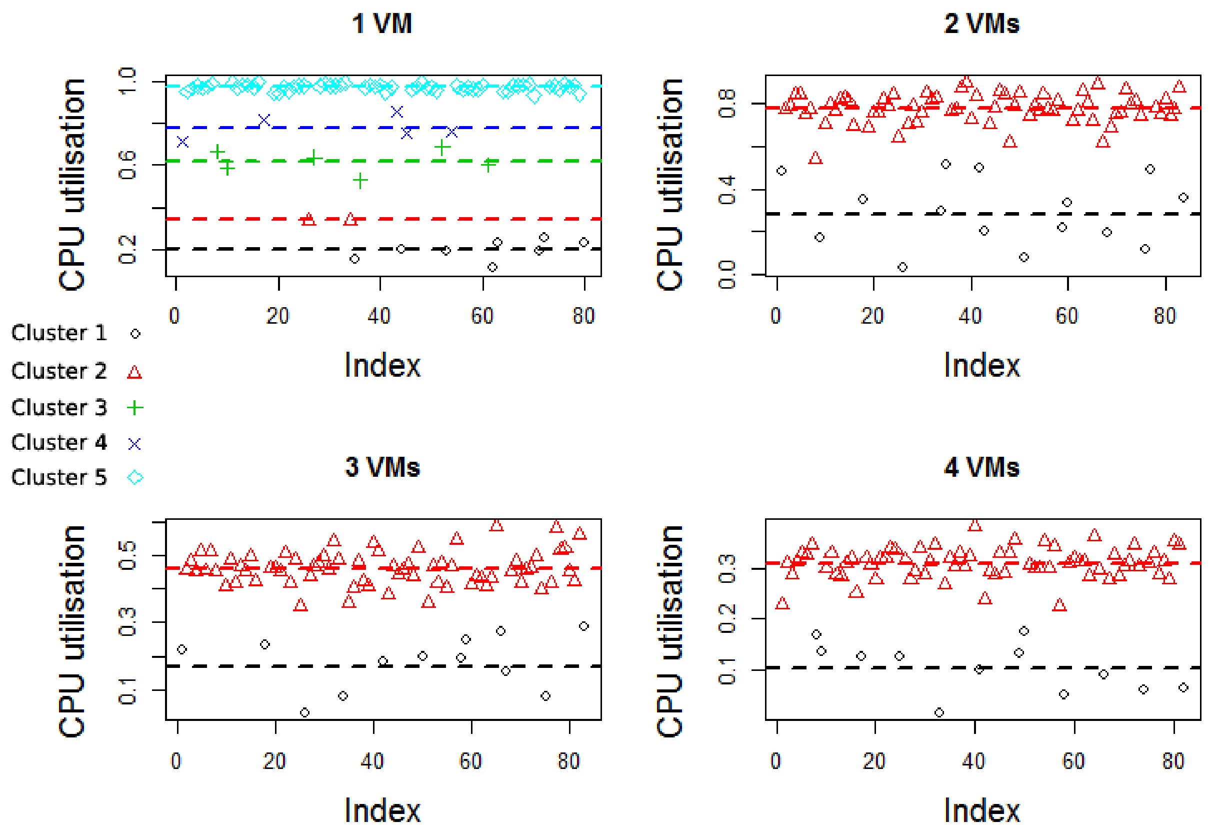


Figure 4.5: Optimal univariate clustering of CPU utilisation per VM number.

scale states, to combine PRISM's *filtering* operation with the temporal operator `X`, to select the set of states which have a high probability to transition to "bad" auto-scale states. That is, we identify the states that occur one time step *before* a "bad" auto-scale state. For our validation purposes we have adopted the former verification strategy (see Section 4.4.7). In Listing 4.6 we show those properties expressed in PRISM.

```
R{bad_auto_scale_state}=? [F end]
P=? [F bad_scale_out=true]
P=? [F bad_scale_in=true]
filter(print, filter(argmax,P=?[X "bad_util_scale_in"]))
filter(print, filter(argmax,P=?[X "bad_util_scale_out"]))
```

Listing 4.6: Properties to be checked

4.4 Model Validation

4.4.1 Validation Methodology

The research question the experiments in this section address is the first part of the main research question which has been phrased in Chapter 1; that is, whether we can use formal verification, and probabilistic model checking in particular, to produce performance guarantees for resource control mechanisms in cloud computing.

Towards addressing the research question, the methodology employed for the design of experiments allows us to investigate the validity of the outputs of the two models constructed for Amazon EC2 and Microsoft Azure against real data obtained from cloud-based applications running on those clouds. Specifically, for the Amazon EC2 case, we investigate how accurate the model is in determining that an auto-scaling policy will result in a QoS violation/non-violation. The performance metrics which determine whether a QoS violation exists or not are CPU utilisation and response time, as discussed in Section 4.3.1. Furthermore, for the Microsoft Azure case, we investigate the accuracy of the model in computing the expected number of "bad" auto-scaling decision taken. The performance metric which determines what constitutes a "bad" auto-scaling decision is CPU utilisation as discussed in detail in Section 4.3.2.

The methodology for the design of the experiments consists of several phases. First, we create the models in PRISM under the various workloads considered and based on the parameters defined in Section 4.2.2. These parameters include the auto-scaling policies in Tables 4.3 and 4.4 for the two public cloud providers. For example, once the model is constructed, for the Amazon EC2 case, we verify whether the auto-scaling policy, given as an input, will result in a QoS violation. This is done through PRISM which computes the probabilities for the properties specified in Section 4.3.1. A similar process is followed for the model developed for Microsoft Azure.

Then, we create experimentation setups on the two public cloud providers, as described in Sections 4.4.2 and 4.4.5. Also, we design load profiles by using a load testing tool (Apache JMeter) and statistical methods to simulate the load patterns, similar to what is described in Sections 4.4.3 and 4.4.6.

Next, we configure the auto-scaling policies on the public cloud providers and generate HTTP requests to the VMs deployed in the public cloud. The same auto-scaling policies are used as inputs both to the PRISM models and to the real auto-scale controllers. Moreover, to minimise the possibility of measurement bias, we perform this experimentation at random time intervals. Then, we obtain the CPU utilisation and response time data for the auto-scaling policies, against which the PRISM models are validated.

To validate the model for the Amazon EC2 case, we employ the ROC methodology [145], which is explained in detail in Section 4.4.4. This is based on statistical methods and allows us to validate our model under a range of validation metrics. For the validation of the Microsoft Azure model, we use the relative error as a validation metric.

In summary, the validation framework consists of three parts for the IaaS case on Amazon EC2 and the PaaS case on Microsoft Azure, respectively. The first is the experimentation setup on the respective public cloud providers (Sections 4.4.2 and 4.4.5), the second is the load profile (Sections 4.4.3 and 4.4.6) and the third is the ROC analysis (Section 4.4.4) for the Amazon EC2 case and the computation of the relative error (Section 4.4.7) for the Microsoft Azure one. Results are discussed for both cases in Section 4.5. Finally, data and other supplementary material from this process are available online [20].

4.4.2 Experimentation Setup on Amazon EC2

First, we discuss the validation process of Amazon EC2, which uses a live public cloud setting: Amazon EC2. The architecture of this experimental setup is illus-

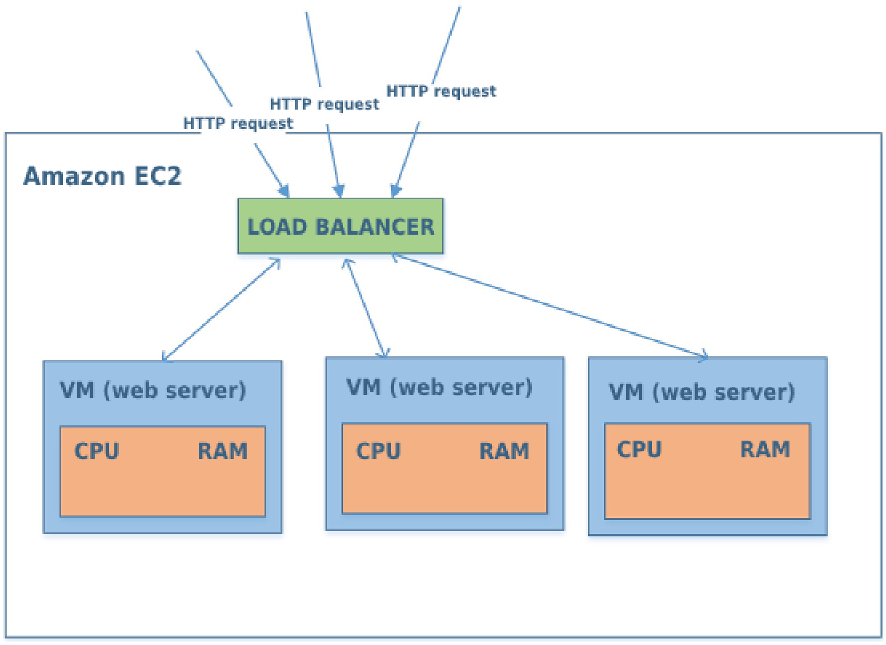


Figure 4.6: Experimental setup on Amazon EC2.

trated in Figure 4.6. We have created an auto-scaling group with minimum and maximum capacity of 1 and 8 VMs respectively. The VM types that were used were *t2.micro* with 1 CPU and 1 GB of RAM. In order to simulate the auto-scaling process in the front layer (web servers) of a cloud-based application, a Python start-up script [146] was launched on those VMs, to simulate the HTTP processing and the CPU consumption. Specifically, the VMs were configured to process each request in 1 second and to send a 500 HTTP response code if it took more than 9 seconds to process the request. Also, an Internet-facing load balancer was used which distributed the load in a round-robin fashion. In addition, to monitor and log all the metrics of the auto-scaling group, we have used Amazon EC2's monitoring service, CloudWatch [147]. The performance metrics are averaged over all the VMs. We monitor and gather the performance metric data for each VM number and for each auto-scaling policy. For each of the policies shown in Table 4.3, we generate load and monitor our VMs on the Amazon EC2 cloud for 10 minutes. Then, we repeat this process, with the same type of load, 30 times for each auto-scaling policy, resulting in 5 hours of data gathering per auto-scaling policy we are validating, approximately.

These samples are then used to validate the verification results of our model.

4.4.3 Load Profile for Amazon EC2

We generate two types of load in the VMs; a periodic and an "aggressive" load pattern. The main reason for choosing a periodic load pattern is because it is considered one of the most representative workload types in cloud computing [148]. To generate a periodic load we have used Apache JMeter [140], which is a professional open source tool for testing web-based applications. Also, on top of Apache JMeter, we make use of the *Ultimate Thread Group* [149] extension for Apache JMeter, which gives us greater flexibility over the threads we are creating. Specifically, we create 3 *Ultimate* thread groups and, within each thread group, we start generating HTTP requests from 1 thread, and then we gradually increase the number of threads. Also, we keep the load duration of each thread for approximately 200 seconds.

The second type of workload we are considering has a greater degree of randomness, and our aim is to validate our model against an aggressive load pattern, but with an inherent degree of randomness. For this type of workload we make the assumption that HTTP requests arrive according to a process with exponentially distributed inter-arrival times.

In order to generate random variables to simulate the workload, the *inverse transform* sampling method was used, which is a sampling method used in performance modelling [150]. This method is relatively simple once the cumulative distribution function (CDF) of the random variable X that is to be generated is known, and the CDF of X can be inverted easily, which holds in our case since the inter-arrival times of the HTTP requests in 1 time unit are exponentially distributed, and the inverse of the CDF of an exponential distribution has a closed-form expression. The algorithm [150] works as follows: 1. Generate $u \in U(0,1)$; 2. Return $X = F_X^{-1}(u)$, where F_X represents the inverse CDF of the exponential distribution. As a result, step 2 will return a realisation of a random variable X from an exponential distribution. In

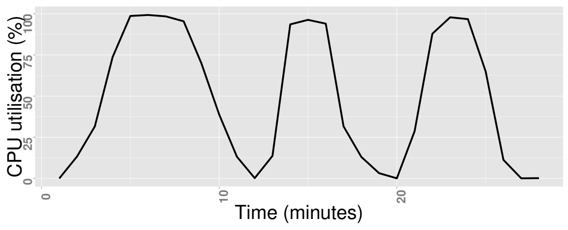


Figure 4.7: Sample CPU utilisation trace under periodic load.

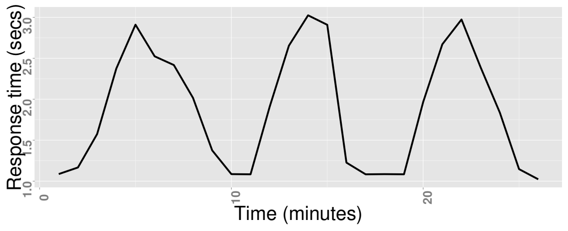


Figure 4.8: Sample response time trace under periodic load.

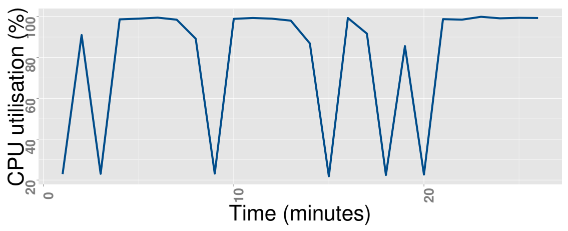


Figure 4.9: Sample CPU utilisation trace under "aggressive" load.

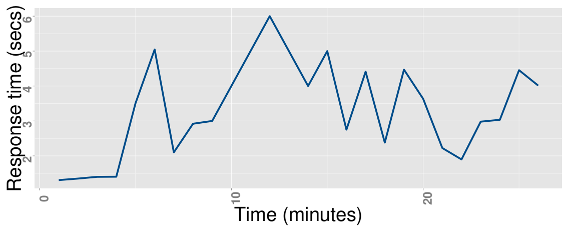


Figure 4.10: Sample response time trace under "aggressive" load.

our case, since we assume 1 time unit, we keep generating exponentially distributed random variables until their sum is 1. We run 1000 simulations and, after storing the results in a vector, we take a sample size of 50 instances. In Figures 4.7–4.10 we show a sample of a CPU utilisation and a response time trace under the two types of workload.

4.4.4 Results and Model Validation via ROC Analysis

In this section, we give an overview of ROC analysis, which is widely used in machine learning and data mining [151], and how it fits our purpose of discriminating between auto-scaling policies that could or could not result in a QoS violation. In a sense, we follow a similar approach to the validation of classification models, by treating the probability as a ranking measure which determines the likelihood of the event of interest, in our case the probability of a QoS violation. ROC can be used to help the decision maker select the appropriate classification threshold, by quantifying the trade-off between sensitivity and specificity. Additionally, it can be used to validate the accuracy of a classification model, by computing the AUC (Area Under Curve) metric, which is one of the most commonly used summary indices [152] .

Our validation starts by ordering the probabilities that have been computed from our PRISM model for each auto-scaling policy. Then, we plot the respective ROC curve which captures all the thresholds between points (0,0) and (1,1) simultaneously. Through this analysis, we are able to find the optimal threshold of discriminating between the auto-scaling policies that could result in a CPU/response time violation. Our criterion of optimality is the point which minimises the Euclidean distance between the ROC curve, and point (0,1), which is often called the point of "perfect classification". Also, this gives us the ability to refine our original violation estimates after we have seen the real data, and to obtain a global threshold for distinguishing between auto-scaling policies. After plotting the ROC curve, we compute the AUC, which in our case can be interpreted as the number of times our model can distinguish performance violations/non-violations of randomly selected auto-scaling policies. This is an "important statistical property" [151] of AUC, and one of the reasons that it has been used so widely for validating the performance of classifiers.

Our PRISM model takes as an input an auto-scaling policy x, and produces a continuous output which is a probabilistic estimate denoting the probability that an auto-scaling policy will result in a QoS violation/non-violation.

Effectively, we wish to find the mapping from x to a discrete variable $y \in \{0, 1\}$, with 0 and 1 indicating the non-violation and violation cases, respectively. However, since the output of the model is continuous and the prediction we want to make is binary, through ROC analysis we find a threshold t such that if $P(x = 1) \geq t$, then we predict that $y = 1$, and if $P(x = 1) < t$, then we predict $y = 0$. We choose t based on our optimality criterion which minimises the Euclidean distance between the ROC curve and point (0,1). Figures 4.11–4.12 show the ROC curves for CPU utilisation and response time under the two load patterns. In Figure 4.11, for example, the threshold t would be approximately 0.8.

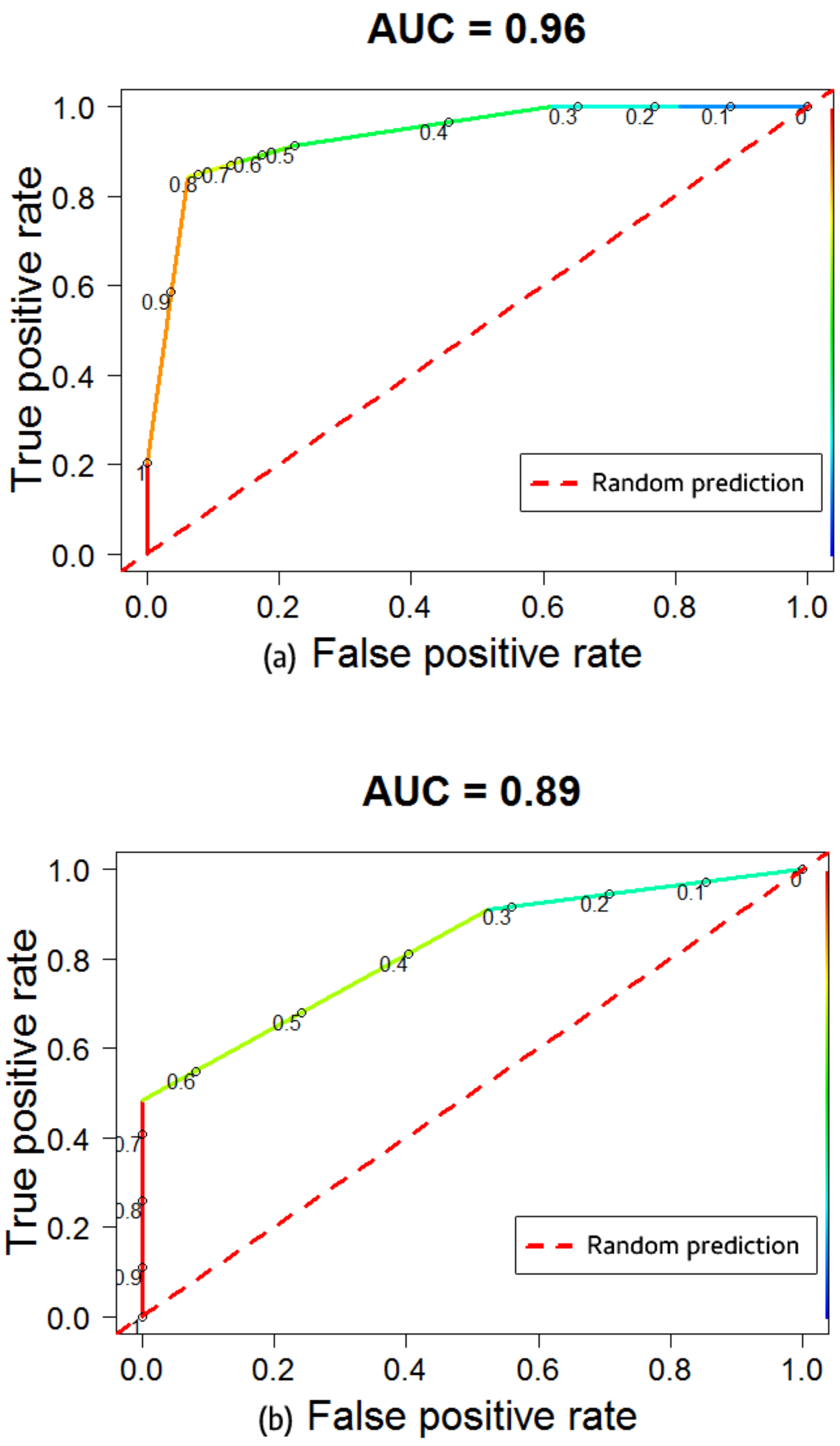
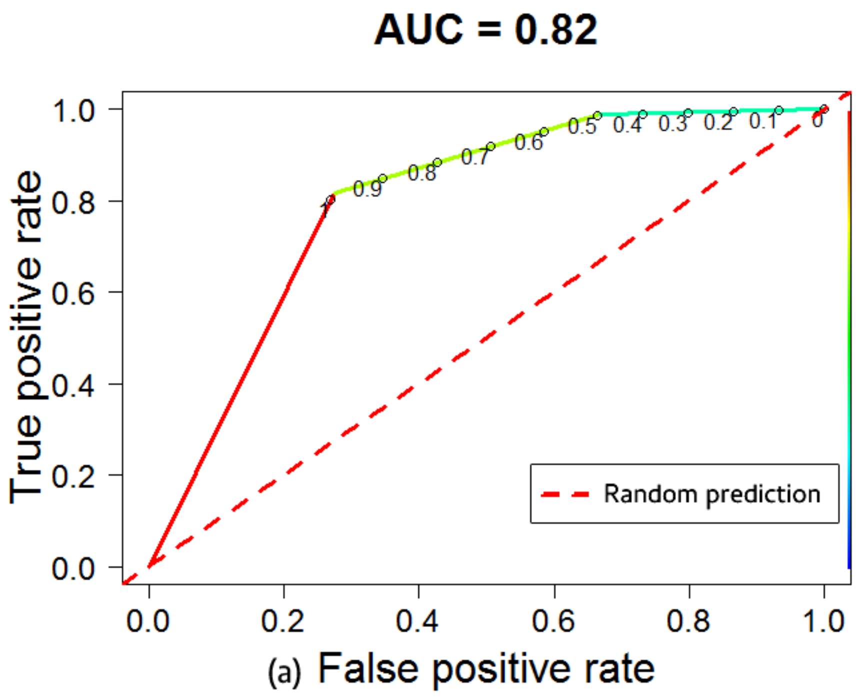


Figure 4.11: ROC curves under periodic load: (a) CPU util. viol.; (b) resp. time.

However, it is important to note that what is considered optimal varies according to the problem one is trying to address, and the relative importance of missing, or increasing true and false positives. This threshold acts now as a global threshold,



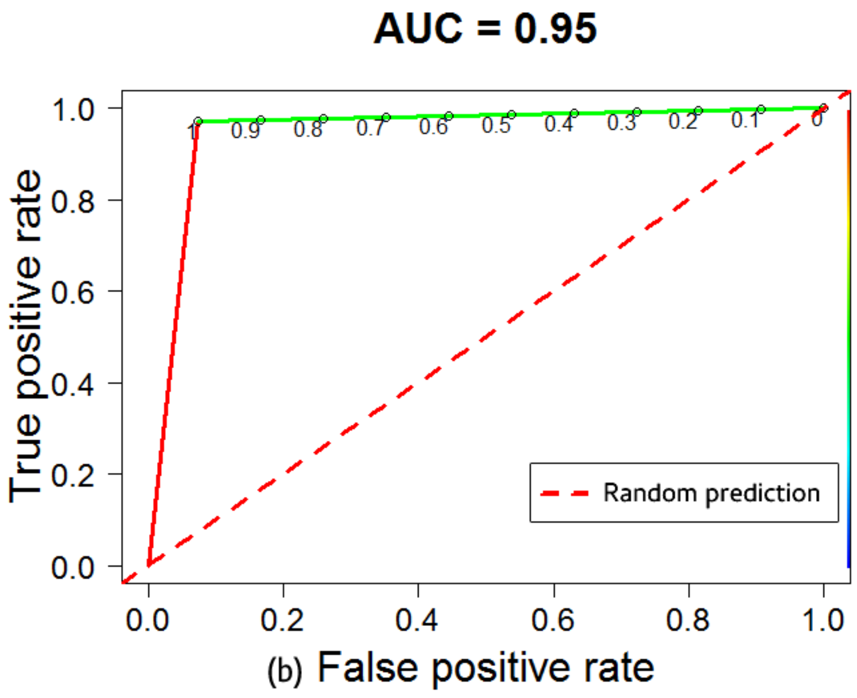


Figure 4.12: ROC curves under "aggressive" load: (a) CPU util. viol.; (b) resp. time.

for the specific auto-scaling policies considered here, and based on that we compute the confusion matrix, and the associated performance measures, where the predicted outcome is determined by this threshold, and the actual values are obtained through real measurements on the Amazon EC2 cloud.

For the ROC curves in Figures 4.11–4.12, we plot the diagonal (red dashed line), which can be thought of as a baseline, which would have been obtained by a random classifier, in order to show how the AUC (Area Under Curve) extends over the diagonal. AUC takes values between 0 and 1, with 1 indicating a “perfect” classifier. For example, if the AUC = 0.5, this is equivalent to a random classification model, and consequently the further the AUC extends over this diagonal, the greater the accuracy of the model.

For completeness, we also consider the MCC (Matthews Correlation Coefficient) [153], despite the fact this is often contrasted with AUC. MCC takes values between -1 and +1, indicating a negative and positive correlation between the predicted and the actual value. Below, we provide its definition, along with several other performance measures used to validate our model, in terms of the metrics from the confusion matrix.

- $ACC = \frac{(TP+TN)}{(TP+FP+FN+TN)}$(overall accuracy of model)
- $TPR = \frac{TP}{(TP+FN)}$ (true positive rate or sensitivity)
- $TNR = \frac{TN}{(FP+TN)}$ (true negative rate or specificity)
- $FPR = \frac{FP}{(FP+TN)}$ (false positive rate, 1 - specificity)
- $FNR = \frac{FN}{(FN+TP)}$ (false negative rate, 1 - sensitivity)
- $MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$

In Tables 4.5 and 4.6, we show the results for these performance measures, and the results are discussed in Section 4.5.

Table 4.5: Performance measures for periodic load

Perf. metrics	ACC	TPR	TNR	FPR	FNR	MCC
CPU util	0.92	0.88	1	0	0.12	0.83
Resp. time	0.9	0.82	0.94	0.06	0.18	0.77

Table 4.6: Performance measures for “aggressive” load

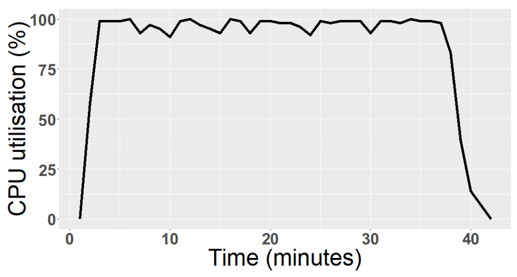
Perf. metrics	ACC	TPR	TNR	FPR	FNR	MCC
CPU util	0.91	0.98	0.65	0.35	0.02	0.7
Resp. time	0.96	0.97	0.93	0.07	0.03	0.86

4.4.5 Experimentation Setup on Microsoft Azure

For the experimentation setup on Microsoft Azure, we have deployed the Bakery web application template to the Azure App Service [154]. This is a template for an online store selling baked goods, developed with ASP.NET and mainly in JavaScript. This allowed us to validate our model under a more realistic and modern web application, while at the same time considering a PaaS cloud computing model (App Service). For this purpose, we have created an App Service Plan, and rented *S1 - Standard* VMs, with 1 core, and 1 GB of RAM. In addition, for our auto-scale plan we considered VMs in the range [1..4] inclusive. The CPU utilisation of the VMs was monitored using Azure's monitoring service on a per-minute basis.

4.4.6 Load Profile for Microsoft Azure

In order to generate load we used four Apache JMeter client machines, deployed on three virtual machines running Ubuntu Linux 16.04 LTS, and one physical machine running Windows 10. The three VMs were deployed through Microsoft Azure to the Western Europe region and they were of type *Standard DS11 v2*, while the physical machine had 4 cores, and 16 GB of RAM. The test scripts were designed in such a way to stress the web application, by performing a sudden increase to nearly 900 threads, and executing a test scenario for approximately 40 minutes. We have also made the XML test scripts available here [20]. In Figure 4.13, we show sample CPU utilisation traces for 1 and 2 VMs, respectively.

(a) 

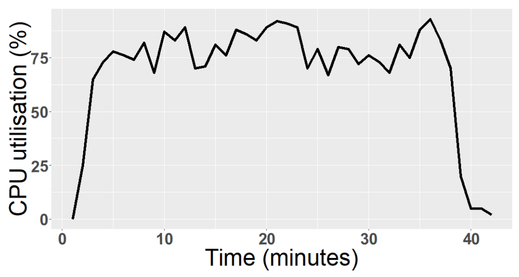
(b) 

Figure 4.13: Sample CPU utilisation traces: (a) 1 VM; (b) 2 VMs.

In order to make our test scenario more interesting, we recorded some browsing patterns and injected random delays between consecutive requests. This allowed our test script to be more representative of a realistic case, by considering the *think time* of users when browsing a web page. We monitor and gather the performance metric data for each VM number and for each auto-scaling policy. For each of the policies shown in Table 4.4, we generate load and monitor our VMs on Microsoft Azure cloud for 40 minutes. Then, we repeat this process 10 times for each auto-scaling policy, resulting in 6.6 hours of data gathering per auto-scaling policy we are validating, approximately. These samples are then used to validate the verification results of our model.

4.4.7 Results and Model Validation

In this section we describe our model validation with respect to what has been discussed in Section 4.3.2. As a metric to compare the measurements from our model with those from the realistic case, we have used the relative error, which has been used in performance modelling and analysis studies [155]. The relative error is defined as $err = \frac{|X_{meas} - X_{sim}|}{X_{sim}}$, where the numerator denotes the absolute error. In Table 4.7, we show the relative error for the expected number of bad auto-scale decisions between our PRISM model, and the measurements taken from our App Service on Microsoft Azure.

Table 4.7: Relative error for "bad" auto-scale actions.

Auto-scale actions	PRISM	Experimental	Cool-down	Rel. error
Scale-out	1.01	1	5 minutes	1%
Scale-in	0.62	0.6	5 minutes	3.33%
Scale-out	2.287	2.8	1 minute	18%
Scale-in	2.8	2	1 minute	40%

4.5 Discussion of Results

In this section we discuss the results for both cloud computing models considered. For the verification of auto-scaling policies on Amazon EC2, our model captures accurately enough the probability of CPU utilisation and response time violations for the specific auto-scaling policies that were shown in the previous section. This can be seen from the AUC values for the auto-scaling policies under the two types of load, as shown in Figures 4.11 and 4.12. For auto-scaling policies which could result in CPU utilisation violation, the AUC value is larger under the periodic load, whereas for policies which could result in response time violations, the AUC is larger for the "aggressive" load. However, for both of these cases, the AUC is high (> 0.8), which shows the high accuracy of our model, under the thresholds [0..1], for the two types of workload.

In Tables 4.5 and 4.6 we show the validation results of the model after selecting a global threshold for each of the four cases. For the auto-scaling policies verified under the periodic load (Table 4.5), we note that the overall accuracy (ACC) is higher for CPU utilisation violation detection, compared to response time. An important observation is that TNR=1, which represents the fact that our model was able to detect, without any error, auto-scaling policies that would not result in CPU utilisation violation, and as a result there were no misclassification of auto-scaling policies which did not cause a CPU utilisation violation in the 10-minute period. Moreover, TPR=0.88 indicates that 88% of the auto-scaling policies which did cause a CPU utilisation violation were correctly identified as such. For the response time, despite the fact that we see a marginal loss of 0.02 compared to the

CPU utilisation in the overall accuracy of the model, we note that TPR and TNR achieve high values of 0.82 and 0.94, respectively. However, we note an increase in the FNR by 0.06.

For the auto-scaling policies validated under the "aggressive" workload (Table 4.6), we note that the overall accuracy of the model with respect to CPU utilisation violation detection remains high (ACC=0.91). Furthermore, the increase in FPR (0.35), compared to the periodic load, means that our model flagged auto-scaling policies as CPU utilisation violators, when in fact they were not. One of the possible causes of this could have been the fact that our gathered CPU utilisation traces contained too many violations, compared to the initial gathered traces that were passed to *k-means*, in order to be used in the state representation of our model. This effect is due to the random nature of the load, and could potentially indicate that more samples are required.

Another observation is the very small value of FNR (0.02), which is considerably more important in our case, since the effects of not flagging an auto-scaling policy as a possible QoS violator could be more serious than the opposite scenario. Finally, for both types of workload, MCC is ≥ 0.7 for both CPU utilisation and response time, indicating a very strong positive relationship between the policies our model flagged as very likely to cause/not cause a violation and the actual outcome when these policies were evaluated in the VMs on Amazon EC2 cloud. We consider the high value of MCC (≥ 0.7) as particularly important since MCC is a balanced measure of the quality of binary classifications, even in cases of imbalanced data.

For the Microsoft Azure case, our high-level goal was to show the importance of the temporal parameters of an auto-scaling policy (e.g. cool-down period) on the performance objectives of an application hosted on the cloud. In addition, we wanted to explore how the non-static parameters of an auto-scaling policy could be modelled and encoded in the PRISM modelling language. With these goals in mind, we have experimented with the auto-scaling policies shown in Table 4.4, and in the next paragraph we report on the results shown in Table 4.7.

For the 5-minute cool-down period in an auto-scaling policy our model captures accurately enough the expected number of bad auto-scale decisions, with the bad scale-out actions being detected more accurately by our model, as the relative error for those actions is 1%. Furthermore, we note that the relative error for the expected number of bad scale-in actions is around 3%. The increase by approximately 2% in the relative error can be explained by the fact that the CPU utilisation traces which were used to validate our model had a slightly greater variability in the scale-in actions for a certain number of VMs.

Also, an important observation that we made by examining our sample CPU utilisation traces is that in most of the cases there existed a pattern with respect to how auto-scale actions are being triggered given the workload used as an input. In addition, from a modelling perspective, the stabilisation period of five minutes after an auto-scale action has been triggered made the correspondence, regarding the duration of each state, between the sampled CPU utilisation traces and our model easier. The negative effect of a decreasing stabilisation period on the accuracy of our model can be seen by the increased relative error for the bad scale-out and scale-in actions. This is because by choosing the minimum cool-down period of one minute we are allowing for the CPU utilisation to stabilise between the consecutive auto-scale requests.

The additional complexity in capturing the temporal pattern of an auto-scale controller which is programmed to fire auto-scale requests consecutively stems from the fact that it becomes a formidable task to detect the actual sending time of an auto-scale request, and the respective time by which the auto-scale request was satisfied. The reasons above provide an explanation regarding the increased relative error under the 1-minute cool-down period. However, we are partially satisfied since our model managed to detect the bad scale-in actions in a moderately effective manner.

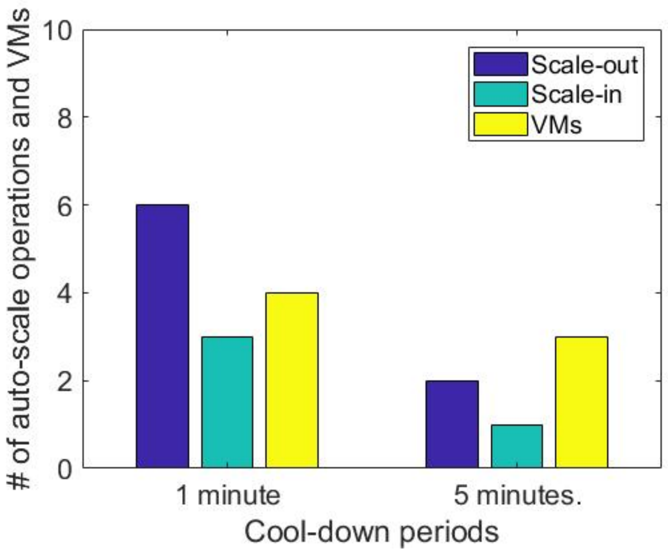


Figure 4.14: Auto-scale operations and VMs under the different cool-down periods.

As part of our exploratory analysis, we show in Figure 4.14 the negative effect of having cool-down periods of short time intervals. For instance, the average number of scale-out operations for the 1-minute cool-down period is 6 compared to 2 for the 5-minute cool-down period. More importantly, the extra scale-out operations did not provide any significant benefit in minimising the QoS violations. In addition, the scale-in actions were 3 under the 1-minute cool-down compared to 1 for the 5-minute cool-down period. We note that the maximum number of VMs which seemed to be required under the 1-minute cool-down was 4 compared to 3 under the 5-minute cool-down.

At this point, we provide an overall assessment of the results in terms of the threats which relate to internal and external validity. Internal validity threats can arise in our case based on how the data, that were used to create and validate the models, were gathered. For the construction of the models, we mitigated such threats by gathering the data which were used to construct the states in our two models at random 30- and 40-minute intervals, respectively. Moreover, we employed well-known clustering methods from the literature to ensure the accuracy of the clustering when creating the models, as it has been discussed in Section 4.2.1. In terms of the data gathered to validate the models, we minimised the possibility of introducing measurement bias by performing this experimentation at random 10- and 40-minute intervals for the two models, respectively.

Furthermore, the data used to validate the models have been made publicly available at [20]. These data include the logs from Amazon EC2's and Microsoft Azure's monitoring services and the associated screenshots of the CPU utilisation graphs, which show whether an auto-scaling policy caused a QoS violation. Note that the CPU utilisation graphs were constructed in real time, at the date of the experiments, from the cloud provider's monitoring services.

The main aim of this work was to ensure its external validity by validating the results on public cloud providers. To this end, for the Amazon EC2 case, we considered two types of workload, while for the Microsoft Azure case we aimed to compensate for having one type of load by injecting random delays between consecutive requests. Also, the main reason for considering a periodic load pattern is that it is deemed one of the most representative in cloud-based applications [156]. However, the data used to create and validate the models were gathered at specific dates in the past, and therefore an assessment of their generalisability would need recent data.

Additionally, for the Amazon EC2 case, regarding the "aggressive" load pattern, we made the assumption that the inter-arrival times of HTTP requests are exponentially distributed. While this is an assumption used in performance analysis studies [150], it is still an approximation of what might occur in a real situation.

Furthermore, while theoretically our models do not depend on workload types, since the user can initialise the CPU utilisation values in the model, we have not performed any experimentation with other types of workloads. Therefore, it is unclear how well the models generalise to other load patterns. Moreover, the models have taken into account specific conditions that relate to the computation of changes in VM capacity and to the prevention of "flapping" situations which are part of the documentation of Amazon EC2 [1] and Microsoft Azure [141], respectively. In this respect, we assume that they cannot represent conditions which might occur in other cloud providers. The experimentation setup could also be extended to include types of VMs other than the ones considered here. Finally, while we assume that the models abstract away information relating to cloud-based applications, they have been validated on the types of applications we consider here.

4.6 Summary of Contributions

The contributions of this chapter can be summarised as follows:

- A novel approach based on performance modelling and formal verification to produce performance guarantees on particular rule-based auto-scaling policies.
- This is, to the best of our knowledge, the first work in which the auto-scaling policies are evaluated using formal verification techniques and which builds probabilistic performance models for horizontal auto-scalers where the models are: i) constructed and validated based on real data gathered from applications deployed on public cloud providers and while the *real* auto-scaler was executing; ii) taking into account aspects (e.g. capacity changes) of the documentation of major public cloud providers (Amazon EC2 and Microsoft Azure). In addition, two cloud computing models (IaaS and PaaS) were considered.
- We have successfully shown that our verification scheme can be of valuable assistance to cloud application owners and system administrators in formally configuring and verifying the auto-scaling policies of their applications/systems in the cloud.

CHAPTER 5

Quantitative Verification of Kalman Filters

In this chapter, we describe our approach to modelling and verification of numerical stability and filter consistency properties of several Kalman filter implementations. This is based on the construction and analysis of a probabilistic model (a discrete-time Markov chain) representing the behaviour of a particular Kalman filter executing in the context of estimating the state of a linear stochastic discrete-time system. The probabilistic model is automatically constructed based on a specification of the filter and the system whose state is trying to estimate. Numerical stability and filter consistency properties are then verified using probabilistic model checking queries. We describe these phases in the following sections.

The remainder of this chapter is structured as follows: In Section 5.1 we describe our formal modelling approach to constructing probabilistic models of Kalman filter execution. Then, in Sections 5.2 and 5.3 we present our novel formal verification techniques for verifying the numerical stability and consistency of the filters, respectively. Next, in Section 5.4 we summarise the main contributions of this chapter. The implementation of the methodology presented in this chapter is shown in Chapter 6, which is then followed by an extensive validation and evaluation process in Chapter 7.

5.1 Constructing Probabilistic Models of Kalman Filter Execution

We define a high-level modelling abstraction which can be instantiated to construct models of various different Kalman filter implementations. The modelling abstraction comprises three components: the first and second correspond to the system and measurement models along with their associated noise distributions; the third is the Kalman filter implementation itself used to estimate the state of the system model in the presence of uncertainty. The first two of these are defined mathematically along the lines described in Section 2.4. The third is specified in detail using a mainstream programming language, since it requires linear algebra data types and operations. Our implementation (see Chapter 6) uses Java and associated numerical libraries.

The discrete-time Markov chain which represents the evolution of the system model along with the filter estimates is not a *static* process. Rather it occurs in a *dynamic* fashion, involving the interaction of several components. For example, we do not assume that the measurements emitted from the system model are already given to us or that the filter estimates are already predetermined. Rather, as the system model evolves from state to state, the Kalman filter executes and tries to estimate its true state, imitating a real-time tracking scenario.

5.1.1 Discrete-time Markov Chain States and Transitions

The variables which define the discrete-time Markov chain's states correspond to the system, measurement and filter models. All of these variables can be made independent of the filter implementations. For example, in a square root filter implementation, P^+ can be either reconstructed from the Cholesky factor C^+, or not by propagating C^+ in each time step, before being passed into the Markov chain's state. This demonstrates the modularity and extensibility of our approach.

The evolution of the states of the Markov chain corresponds to the system model perturbed by different noise values. Each of the Markov chain's states stores the "true" values of the system model's state and the noisy measurements emitted at each time step k. These variables, along with the a posteriori state estimate and the estimation-error covariance, are stored in the Markov chain state, because they are needed for verification purposes. Then, before the Markov chain transitions to the next state (between time k and $k+1$), the time update of the corresponding filter variant is invoked. Both a priori variables depend on their a posteriori counterparts.

Specifically, once we are in a state for time instant k, our goal is to compute in the next state at time $k+1$ both the system model's updated state vector and the a posteriori variables of the respective filter, $\hat{x}^+, P^+$. The a priori variables of the Kalman filter types are encapsulated between these two updates as an intermediate step. Note that $\hat{x}$ and P are essentially the same variables which are used in the computation of both the a priori and a posteriori state estimate and estimation-error covariance matrix, respectively. What distinguishes $\hat{x}$'s semantics is whether the measurement z has been processed. This allows us to concretely define the notion of time k in each of the Markov chain's states.

In particular, a time instant k in the Markov chain can be thought of encompassing i) state variables *before* the measurement is processed and ii) state variables *after* the measurement has been processed. Combining this temporal order into one state allows us to save storage by merging what would otherwise require two states to be represented.

5.1.2 Noise Model Discretisation

Transitions in the discrete-time Markov chain capture the stochastic evolution of the Kalman filter due to noise in the system and measurement models. Since the latter are continuous probability distributions, we need to discretise these to model them in the Markov chain, which allows us to obtain reasonable approximations of the effect of the noise.

The number of outgoing transitions from every state and their probability values are determined by a *granularity level* of the noise, that we denote `gLevel`. The Gaussian distribution of the noise is discretised into `gLevel` disjoint intervals. The intervals used for each granularity level are shown in Table 5.1. In determining the transition probabilities and the noise values that influence the system model's state vector, the following problem had to be addressed: The system model is a stochastic linear dynamical system perturbed by Gaussian noise which means that we are dealing with a continuous random variable which takes values over a continuous range, and its distribution is described by a probability density function (PDF). On the other hand, the probabilistic model is a discrete-time Markov chain which captures the evolution of the system over time. This means that from our modelling abstraction's viewpoint we are dealing with a discrete random variable which takes values from a discrete set, and its distribution is described by a probability mass function (PMF). The latter implies that an *exact* probability is assigned to each of those values. In summary, the problem can be stated as finding a discrete approximation of the Gaussian distribution, however, this raises the following questions:

1. How to compute the probability values for each of the transitions of the Markov chain based on the granularity level, given as input?

2. How to compute the expected value of the interval which has been truncated, especially for those cases in which the mean of the distribution is not included in the interval?

Table 5.1: Intervals according to the granularity level.

gLevel	**Intervals**
2	$[-\infty..\mu]$, $[\mu..+\infty]$
3	$[-\infty..-2\sigma]$, $[-2\sigma..+2\sigma]$, $[+2\sigma..+\infty]$
4	$[-\infty..-2\sigma]$, $[-2\sigma..\mu]$, $[\mu..+2\sigma]$, $[+2\sigma..+\infty]$
5	$[-\infty..-2\sigma]$, $[-2\sigma..-\sigma]$, $[-\sigma..+\sigma]$, $[+\sigma..+2\sigma]$, $[+2\sigma..+\infty]$
6	$[-\infty..-2\sigma]$, $[-2\sigma..-\sigma]$, $[-\sigma..\mu]$, $[\mu..+\sigma]$, $[+\sigma..+2\sigma]$, $[+2\sigma..+\infty]$

Before explaining our approach for dealing with the questions above it is worth mentioning how the intervals are created in the first place. In Table 5.1 we show the intervals for which each granularity level corresponds to. The measure used to determine these intervals is the standard deviation, σ, which is a common practice in statistical contexts; see for example the so-called $68-95-99.7$ rule which states that assuming the data are normally distributed then 68%, 95% and 99.7% of them will fall between one, two and three standard deviations of the mean, respectively. This statement can be expressed probabilistically as well by calculating the cumulative distribution function (CDF) of a normally distributed random variable X, usually by converting it to its *standard* counterpart and using the so-called standard tables.

While computing the probability that a noise value will fall inside an interval is relatively easy, the computation of its expected value is slightly more difficult. This is because we can choose to either truncate the distribution to intervals which contain the mean value of the distribution, which is the easier case, or to intervals which do not. See for instance Figure 5.1 and assume that the intervals from $[-2\sigma..+2\sigma]$, and $[-2\sigma..\mu]$ have been chosen. Note that we represent the intervals as *closed* with the symbols [..] which denote that the upper and lower limits are included. However, as we are dealing with a continuous distribution, having chosen to represent them as *open* intervals it would have been equally correct since, for example, $P(X = \mu) = 0$. For the first case, the expected value will be 0, which is the mean of distribution; for the second, this is not true. Usually for those cases one might use a simple heuristic such as dividing the sum of the two endpoints of the interval by two which is actually quite common. However, it might not be representative of the actual mean, since it

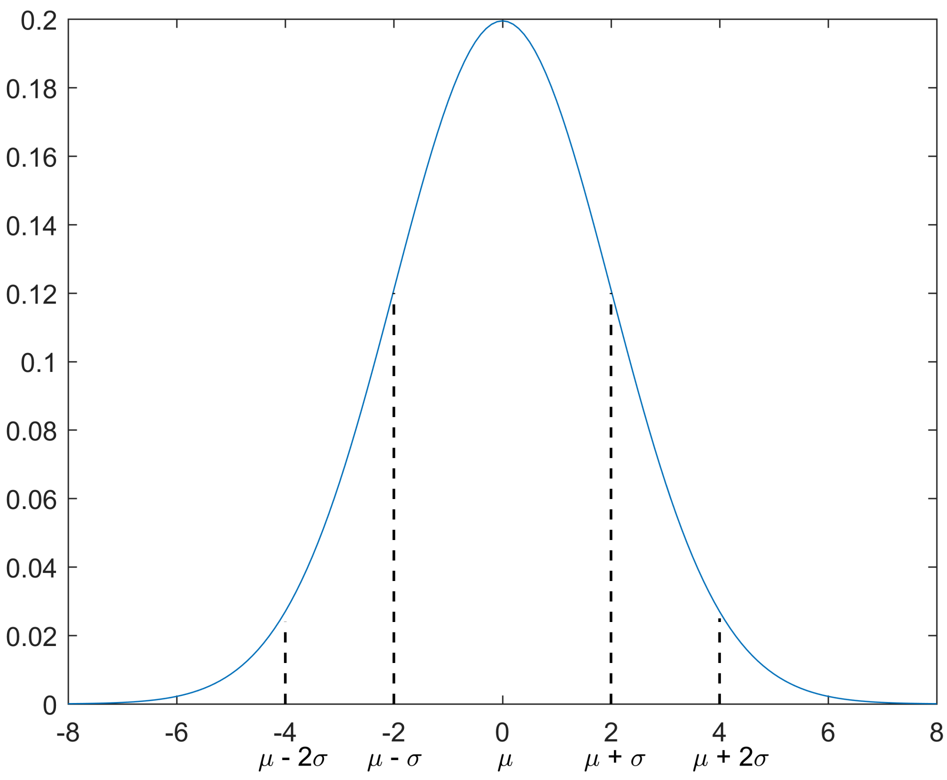


Figure 5.1: Gaussian distribution with $\mu = 0$ and $\sigma = 2$.

does not weigh the values lying inside the interval according to the corresponding value of the density correctly. In other words, since the mean is also interpreted as the *"centre of gravity"* of the distribution [52], in the case of truncated intervals which do not contain the mean, more accurate techniques are needed. The following example demonstrates this reasoning.

Example 5.1. Consider the case where the distribution in Figure 5.1 has been truncated with a granularity of level 6, according to Table 5.1. Assume that we want to compute the mean of the interval $[+\sigma.. + 2\sigma]$, which for the distribution in the figure amounts to [2..4]. If someone uses the simple common heuristic to compute the mean given by $\frac{2+4}{2}$ then the mean will be equal to 3. However, one can note by visually inspecting the figure that this is not a very accurate approximation, because a uniform weight of 0.5 has been assigned to the two values. Consequently, the resulting value, the mean, will be in the middle of the interval, maintaining an equal distance from the upper and lower endpoints of the interval. In fact, the correct mean is 2.77, which is closer to the lower endpoint of the interval.

The probabilities of the Markov chain for a given granularity level are computed by first standardising the random variable, the noise in our case, and then evaluating its CDF at the two endpoints of the corresponding interval. Then, by subtracting them, we obtain the probability that it will fall within a certain interval. The way a normal random variable X is standardised is by calculating the so-called *Z score* given by $Z = \frac{X-\mu}{\sigma}$, which measures the distance between X and the distribution mean μ in standard deviations. Note that in our case $\mu = 0$ and $\sigma^2 = 1$, since X is a standard normal random variable. The CDF of a standard normal random variable X, denoted as Φ, is given as follows [52]:

$$\Phi(x) = P(X \leq x) = P(X < x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt \tag{5.1}$$

Once the probabilities have been computed, it remains to find the expected value of the random variable for the corresponding intervals. In order to avoid the situation described earlier, in Example 5.1, and obtain the mean in a more accurate way, we have used the *truncated normal distribution* to compute the mean for the respective intervals. Formally, if a normal random variable X is normally distributed and lies within an interval $[a..b]$, where $-\infty \leq a \leq b \leq +\infty$, then X conditioned on $a < X < b$ has a truncated normal distribution. There are four different ways to truncate a normal distribution [157, 158]. These four cases are described below:

1. $a = -\infty$ and $b = +\infty$. No truncation. The resulting distribution is the original normal distribution.
2. $a > -\infty$ and $b = +\infty$. Singly truncated from below.
3. $a = -\infty$ and $b < +\infty$. Singly truncated from above.
4. $a > -\infty$ and $b < +\infty$. Doubly truncated, which amounts to the type of the truncations we perform.

The probability density function of a normally truncated random variable X is characterised by four parameters: i-ii) the mean μ and standard deviation σ of the *original* distribution and iii-iv) the lower and upper truncation points, a and b. Formally, it is defined as follows:

$$\psi(x;\mu,\sigma,a,b) = \frac{\phi(\frac{x-\mu}{\sigma})}{\sigma\Big(\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})\Big)} \tag{5.2}$$

Compactly, the mean value of the noise for a corresponding interval can be expressed as the conditional mean, $E[X|a < X < b]$, given by the following formula [157]:

$$E[X|a < X < b] = \mu + \sigma\frac{\phi(\frac{a-\mu}{\sigma}) - \phi(\frac{b-\mu}{\sigma})}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})} \tag{5.3}$$

Note that in the expression above, ϕ and Φ denote the probability density and the cumulative distribution function, of the standard normal distribution, respectively. As a result, the computation of the transition probabilities and the mean values for each of the corresponding interval can be done in a unified manner. The example below demonstrates what has been discussed so far for an interval, and a flow diagram of the whole process is shown in Figure 5.2.

Example 5.2. Let us assume that a normal random variable is passed as input with a normal distribution as shown in Figure 5.1, with $\mu = 0$ and $\sigma = 2$, and assume that a granularity level of 3 has been given as input. The first step is to construct the *unstandardised*, denoted as truncation points according to Table 5.1. Next, the truncation points are standardised. Here, we show this process for the interval $[-2\sigma .. + 2\sigma]$.

This results in -2 and 2 for the standardised lower, ($a2\_std$), and upper truncation, ($b2\_std$), points, respectively. Next, we compute the probability that a random

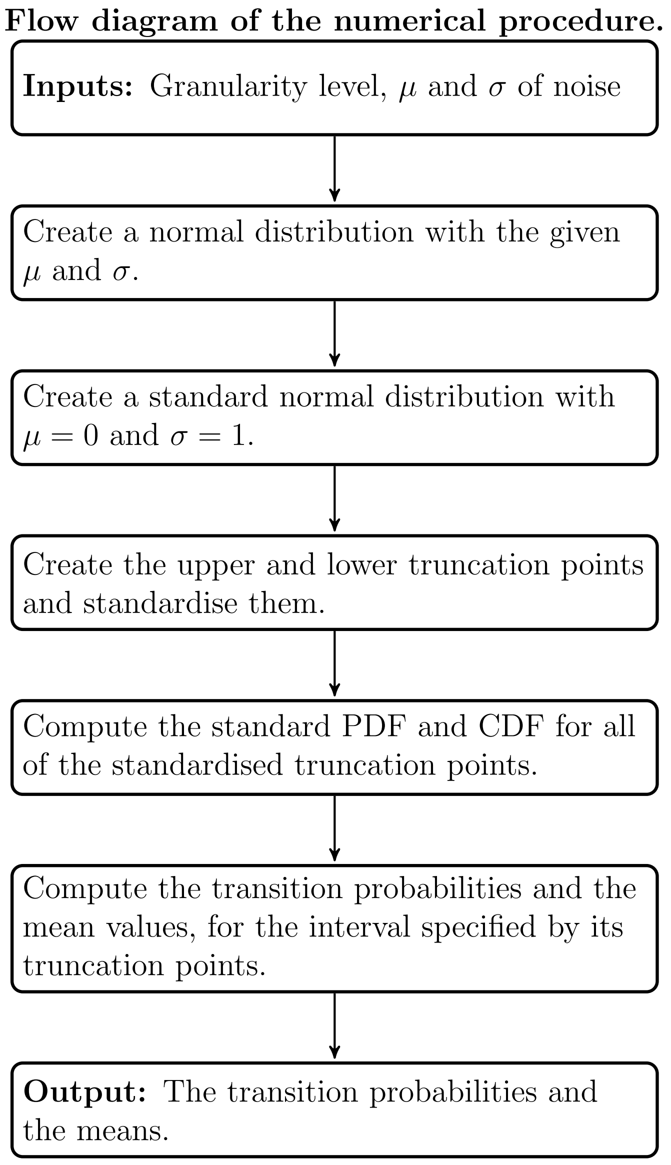

Flow diagram of the numerical procedure.
Inputs: Granularity level, μ and σ of noise
Create a normal distribution with the given μ and σ.
Create a standard normal distribution with μ = 0 and σ = 1.
Create the upper and lower truncation points and standardise them.
Compute the standard PDF and CDF for all of the standardised truncation points.
Compute the transition probabilities and the mean values, for the interval specified by its truncation points.
Output: The transition probabilities and the means.

Figure 5.2: Flow diagram of the numerical procedure to create transition probabilities and the means for the noise model, which perturb the system model.

variable (e.g. noise) will fall inside this interval.

$$P(a2\_std \le X \le b2\_std) = \Phi(X < b2\_std) - \Phi(X < a2\_std) \approx 0.954 \tag{5.4}$$

For the computation of the mean in this case, the result, which is zero, is already known to us, because it is contained in the interval considered. However, for the sake of this example it is also computed. This is expressed as the conditional mean

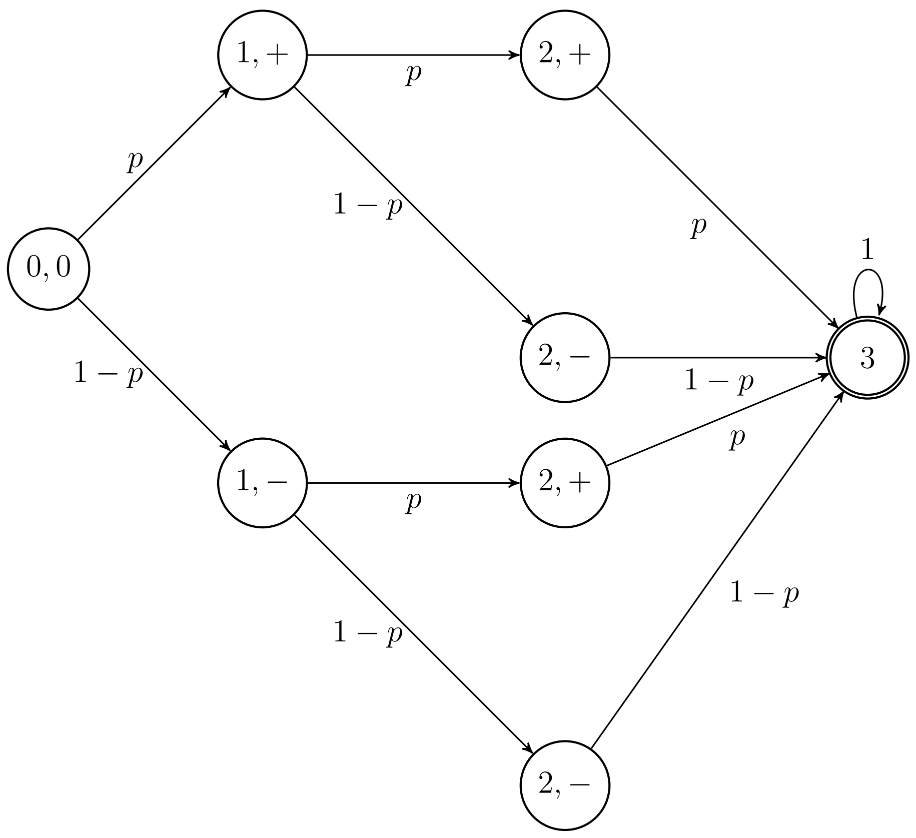


Figure 5.3: A schematic representation of the Markov chain model with granularity level 2.

of X given that it can only take values inside this interval, using the formula:

$$E[X|a < X < b] = \mu + \sigma \frac{\phi(\frac{a-\mu}{\sigma}) - (\frac{b-\mu}{\sigma})}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})} \tag{5.5}$$

$$E[X|a < X < b] = 0 + 2\frac{0.05 - 0.05}{0.977 - 0.022} = 0 \tag{5.6}$$

In Figure 5.3 we show a simplified visual representation of the Markov chain with a granularity level of 2. The numbers inside each state correspond to the discrete time instants k. The $+, -$ signs correspond to whether the noise perturbing the system model is positive or negative. Similar to what has been described before, if we consider `gLevel` intervals in which the mean value of the noise is included, the Markov chain transitions to a state with probability p where the mean value of the noise is positive and is computed similar to the procedure explained in Section 5.1.2. Symmetrically, with probability $1 - p$ the Markov chain transitions to a state where the noise takes a negative value. The terminal state, state 3, of the Markov chain is also called an *absorbing* state since once entered there is no outgoing transition. In our modelling abstraction absorbing states are determined based on the value of the `maxTime` variable, which is given as an input, and determines the maximum time the model runs.

It should be noted that the discretisation and truncation of the stochastic noise model does not affect the validity of the verification results of the numerical properties we describe in Section 5.2. This is because these properties, and numerical errors in Kalman filters in general, mostly relate to numerical instabilities in the the computation of the estimation-error covariance matrix P, because this matrix is independent of the measurements emitted from the system model.

On the other hand, the discretisation and truncation of the stochastic noise model may affect the validity of the verification results, by decreasing their accuracy, for the properties which relate to modelling errors (Section 5.3). This is because we assume that the highest value of the granularity level of the noise (`gLevel`) considered (i.e. 6), provides the most accurate approximation of the noise. Consequently, since the verification of these properties depends on the measurements emitted from the stochastic noise model, a loss in the accuracy of the verification result may be expected.

Therefore, a potential user of the approach should conduct a trade-off analysis in order to determine an appropriate `gLevel` value, to verify the properties related to modelling errors (Section 5.3). However, when verifying the numerical properties shown in Section 5.2, the `gLevel` value should be set to the minimum value possible (e.g. 2).

5.2 Verification of Numerical Stability

Next, we discuss how to capture numerical stability properties for our Kalman filter models (see the earlier summary in Section 3.5.1) using the probabilistic temporal logic [21] of the PRISM model checker [19]. We explain the properties below, as we introduce them, and refer the reader to [21] for full details of the logic. From this analysis the steady-state filter is excluded, since its steady-state covariance is computed prior to the execution of the filter, and as a result its numerical robustness can be determined beforehand.

5.2.1 Verification of Numerical Stability of the Conventional Kalman Filter

Verifying symmetry of the estimation-error covariance matrix. The symmetry of the estimation-error covariance matrix is an invariant which needs to hold across every path of the Markov chain. It is worth noting, however, that verifying invariants does not absolutely require the use of a model checker, since all the states are generated and can be checked as the generation process is carried out. To construct this property we use an atomic proposition *isSymmetric* for the states in which the absolute difference between the corresponding elements of P^+ which are not on the main diagonal, is close to zero within some predefined tolerance, usually the so-called machine epsilon.

In order to specify the probability that the estimation-error covariance matrix remains symmetric, we use the formula $\mathtt{P}_{=?}[\ \mathtt{G}\ \mathit{isSymmetric}\,]$, where $\mathtt{P}$ is PRISM's probabilistic operator. Also, the temporal logic operator $\mathtt{G}$, which is often referred to as "always" or "globally", is used to represent invariance. If the result of this probabilistic query returns a probability value less than 1, then this means that P^+ is not a symmetric matrix and consequently cannot be considered a valid covariance matrix. As a result the Kalman filter considered, should be rejected.

Verifying positivity of the diagonal elements of the estimation-error covariance matrix. This is a "simple" property which relates to verifying where P^+ remains positive definite throughout the execution of the filter, or not. "Simple" here is used to denote the easiness this property can be constructed, in order for "theoretical impossibilities", such as negative variances, to be checked. However, this property should be used with cautiousness in terms of determining whether P^+ is positive definite or not. This is because, it is well-known that in a positive definite matrix its diagonal elements are positive, however the converse is not always true. For example, the diagonal elements of a matrix can be positive but the matrix itself need not be positive definite. An atomic proposition *isDiagPos* is used for the states in which the diagonal elements of P^+ are positive, and the model is then queried using the formula $\mathtt{P}_{=?}[\,\mathtt{G}\ \mathit{isDiagPos}\,]$. If the query returns a probability value < 1, then this means that P^+ cannot be considered a valid covariance matrix, and consequently the Kalman filter should be rejected.

Verifying positive definiteness. In order to construct this property, we perform an eigenvalue-eigenvector decomposition of P^+ into the matrices $[V, D]$. The eigenvalues are obtained from the diagonal matrix D, and their positivity is determined and used to labelling each state of the Markov chain accordingly: we use an atomic proposition *isPD* for states in which P^+ is positive definite.

We can then specify the probability that the matrix remains positive definite for the duration of execution of the filter using formula $\mathtt{P}_{=?}[\,\mathtt{G}\ \mathit{isPD}\,]$, and reject Kalman filters for which *isPD* is not satisfied at all states of the model.

Verifying positive semidefiniteness. In many applications it is useful to determine those states in which P^+ is positive semidefinite. This is because not only a positive semidefinite matrix can be considered "troublesome" in the context of Kalman filtering and estimation problems in general, but also because of the extent to which a positive semidefinite matrix is susceptible to numerical errors, that it might become indefinite. This property follows a similar construction process to the previous property; first an eigenvalue-eigenvector decomposition of P^+ into the matrices $[V, D]$ is performed. The eigenvalues are obtained from the diagonal matrix D, and their nonnegativity is determined and used to labelling each state of the Markov chain. An atomic proposition *isPSD* is used for the states P^+ is positive semidefinite.

Then, the formula $\mathtt{P}_{=?}[\ \mathtt{G}\ \mathit{isPSD}\ ]$ is specified and based on the result, the Kalman filter will be considered as valid or invalid.

Examining the condition number of the estimation-error covariance matrix. The verification of certain numerical properties, such as those related to positive definiteness, is a challenging task and should be treated with caution. This is because, while convenient, focusing the verification on whether an event will occur or not, might not capture inherent numerical difficulties related to the numerical stability of state estimation algorithms. In other words, it does not suffice to check whether P^+ is positive definite or not by checking its eigenvalues because, as mentioned earlier, if they are in close proximity to zero, then round-off errors could cause them to become negative [14].

For example, it is often the case that estimation practitioners want to detect matrices that are close to becoming *singular*, a concept which is often referred to as *"detecting near singularity"* [60]. In other words, since a positive definite matrix is nonsingular, one wants to determine the "goodness" of P^+ in terms of its "closeness" to singularity, within some level of tolerance, usually the machine precision [14]. A matrix is said to be *well-conditioned* if it is "far" from singularity, while *ill-conditioned* describes the opposite. In order to quantify the goodness of P^+, we use the so-called *condition number*, which is a concept used in numerical linear algebra to provide an indication of the sensitivity of the solution of a linear equation (e.g. $Ax = b$), with respect to perturbations in b [14, 13]. In our case, this concept is used to obtain a measure of goodness of P^+.

The condition number of P^+ is given as $\kappa(P^+) = \sigma_{max}/\sigma_{min}$, where σ_{max} and

σ_{min} are the maximum and minimum singular values, respectively [16, 13]. These can be obtained by performing the singular value decomposition (SVD) of P^+. A "small" condition number indicates that the matrix is well-conditioned and nonsingular, while a "large" condition number indicates the exact opposite. Note that the smallest condition number is 1 when $\sigma_{max} = \sigma_{min}$.

We express this property as the formula $\mathtt{R}^{cond}_{=?}[\,\mathtt{I}^{=k}\,]$, which gives the expected value of the condition number after k time steps. We assign the condition number to each state of the discrete-time Markov chain using a reward function *cond* and we set k to be `maxTime`, the period of time for which we verify the respective Kalman filter variant.

Examining the reciprocal of the condition number of the covariance matrix. Sometimes it is more intuitive to compute the reciprocal of the condition number. This allows us to define it in the range of numbers between 0 and 1, and relate it more "naturally" to a measure of goodness, in a "larger the better" sense. In this case a matrix whose inverse condition number is close to 1 is considered a well-conditioned matrix, while a matrix with an inverse condition number near 0 is an ill-conditioned matrix. The reciprocal of the condition number is computed by dividing the smallest by the largest eigenvalue, or by diving 1 by the condition number κ (e.g. $1/\kappa(P)$). Similar to the previous property, this is a reward-based property whose purpose is to assign a measure of goodness of P^+ to the Markov chain states for further analysis.

It is expressed as the formula $\mathtt{R}^{invCond}_{=?}[\,\mathtt{I}^{=k}\,]$, which gives the expected value of the inverse of the condition number after k time steps. The reward function *invCond* serves a similar purpose to the reward function *cond*, defined previously.

Providing bounds on numerical errors. Another useful aspect of the condition number is that it can be used to obtain an estimate of the precision loss that numerical computations could cause to P^+. For instance, for a single precision and a double precision floating-point number format, the precision is about 7 and 16 decimal digits, respectively. Since our computations take place in the decimal number system, the logarithm of the condition number (e.g. $log_{10}(\kappa(P^{+})))$, gives us the ability to define more concretely when a condition number will be considered "large" or "small" [13, 15, 41]. For example, a $log_{10}(\kappa(P^{+})) > 6$ and a $log_{10}(\kappa(P^{+})) > 15$ could cause numerical problems in the covariance computation and render P^{+} as ill-conditioned when implemented in a single and a double precision floating-point number format, respectively.

So, to verify this property we construct a closed interval whose endpoints will be based on the appropriate values of the numerical quantity of $log_{10}(\kappa(P^{+})$. This lets us label states whose $log_{10}(\kappa(P^{+}))$ value will fall within "acceptable" values in the interval, when, for instance, double precision is used. We compute, in a similar fashion to the first property above, the probability $\mathtt{P}_{=?}[\ \mathtt{G}\ \mathit{isCondWithin}\,]$, where *isCondWithin* labels the "acceptable" states. A probability value of less than 1 should raise an alarm that numerical errors may be encountered.

5.2.2 Verification of Numerical Stability of the Carlson-Schmidt Square-Root Filter

There are two ways one can define properties for the Carlson-Schmidt square-root filter. One might be interested in the estimation-error covariance matrix per se by reconstructing it, in every or in a specific time step k, in order to analyse it further. However, this approach might be of interest from a statistical point of view; for example when the goal of the quantitative verification is to explore possible modelling errors. Furthermore, adopting this approach for the verification of numerical properties of the Carlson-Schmidt square-root filter is not ideal, since it might not have made apparent the numerical robustness of square-root filters. Moreover, it would have resulted in the unnecessary replication of the numerical properties which were used for the verification of the conventional Kalman filter. For instance, the properties which are related to the verification of symmetry and positive semidefiniteness of the estimation-error covariance matrix would be considered redundant, since the square-root filter results in an estimation-error covariance matrix, which is symmetric positive semidefinite *by construction* [54, 56]. As a result, the proposed properties related to the examination of the condition number of the "square-root" of P^+ are evaluated on its Cholesky factor, C^+, which is propagated forward in time.

5.2.3 Verification of Numerical Stability of the Bierman-Thornton UD Filter

Verifying positive definiteness. In a similar fashion to the Carlson-Schmidt square-root filter, the UD filter exhibits the same nice numerical properties such as ensuring symmetry and positive semidefiniteness by construction. Furthermore, we can either choose to reconstruct the estimation-error covariance matrix in every time step or process the U and D factors only. To check whether P^+ is positive definite it suffices to label the states of the Markov chain according to whether the elements of the diagonal matrix D are greater than zero, or not.

5.3 Verification of Modelling Error Compensation Techniques

In this section, we discuss how to perform a rigorous quantitative analysis of the effectiveness of the Kalman filter variants, in particular tackling the detection of the divergence of the filter in the presence of modelling errors. Quantitative verification is used to verify the so-called consistency properties of Kalman filters. This can be used to drive the so-called filter tuning process, which optimises parameters of the filter, and produces guarantees on the chosen parameter values.

Verifying that the innovation is bounded by its variance. In the first statistical test we verify that the fraction of times the innovations fall within two standard deviations ($\approx 1.96\sqrt{S}$) of the mean is at least 95%. In order to express this property as a temporal logic formula, we use a reward function *inRange*, which assigns a reward of 1 to the states in which the value of the innovation y_k falls within two standard deviations, and 0 otherwise.

Then, the reward-based property which verifies that 95% of the times the *inRange* reward falls within two standard deviations, is expressed as a formula of the form the $\mathtt{R}^{inRange}_{=?}[\,\mathtt{C}^{\leq maxTime}\,]$. The property $\mathtt{C}^{\leq maxTime}$ is called a cumulative reward and associates the *inRange* reward to each path of the Markov chain. The result of the reward is then compared to a *bound*, which is determined upon the maximum time the model runs. This bound is computed as the product of 0.95 with the `maxTime` variable, taking into consideration that measurements are processed as scalars. For instance, if the model runs for 20 time steps, the bound on the reward will be 19 (0.95×20), since $19/20 \approx 95\%$.

In our case, we assume that there are no measurements available during the first two time steps. The reason is due to the way we index time, starting from 0, since the first two states are used in the initialisation of the Markov chain. This means that when the model runs for 20 time steps (i.e. `maxTime`=20), the bound which should be exceeded by the result of the *inRange* reward, is ≈ 17. As a result, if the result of the formula is a value less than 17, then this is an indication that the Kalman filter is inconsistent.

Verifying that the magnitude of the innovations should be proportionate to the covariance. The second statistical test we consider revolves around the concept of Hypothesis Testing, in which the *null* hypothesis H_0 describes the *status quo* or the default theory [159], and the *alternative* hypothesis H_1 the exact opposite. The overall concept of Hypothesis Testing is to test H_0 against H_1 and decide whether to reject H_0, based on the "amount" of evidence we acquire from the data, which in our case corresponds to the measurements emitted at each time step. It is important to note that there does not exist any symmetry between H_0 and H_1, meaning that failure to obtain the required evidence, does not imply that H_0 is *true*, rather than the fact that enough evidence has not been acquired to reject it. In our context, the null hypothesis, H_0, describes the hypothesis that the Kalman filter is consistent, or more specifically that the magnitude of the innovations y_k is proportionate to the covariance computed by the Kalman filter. Furthermore, the alternative hypothesis, H_1, describes the hypothesis that the Kalman filter is inconsistent.

The *statistic* which we structure the statistical test around, is the so-called *normalised innovation squared* (NIS) [23], given as: $\epsilon^2_{y_k} = y_k^{\intercal} S^{-1} y_k$. Formally, a statistic is a function of the data, and since it is a random variable it has a certain distribution [159]. Note that $\epsilon^2_{y_k}$ conforms totally to the formal definition of a statistic, since $\epsilon^2_{y_k}$ is a random variable, it can be expanded to $\epsilon^2_{y_k} = \left(z_k - H\hat{x}_k^-\right)^T \left[HP_k^- H^T + R_k\right] \left(z_k - H\hat{x}_k^-\right)$, and it follows a χ^2 distribution. However, since the information at a particular time instant k might not be sufficient to draw reliable conclusions, [23] proposed that the average of K time steps should be taken into account. Therefore, the statistic we consider is the *time-average normalised innovation squared*, given by $\bar{\epsilon}^2_{y_k} = \frac{1}{K}\sum_{i=1}^{K} \epsilon^2_{y_k}$.

Under the assumption that H_0 has been retained, or in other words that there is no sufficient evidence to declare the Kalman filter as inconsistent, $K\bar{\epsilon}^2_{y_k}$ is χ^2 distributed with Kn degrees of freedom, where n is the dimension of the measurement vector [23] [160]. Note that the χ^2 distribution, unlike the Gaussian distribution, is indexed by one integer parameter, the degrees of freedom. We accept H_0 as long as $\bar{\epsilon}^2_{y_k} \in [r1, r2]$, where the confidence region is defined as $P\left\{\bar{\epsilon}^2_{y_k} \in [r1, r2] \mid H_0\right\} = 1 - \alpha$, and the significance *level* α is set to 0.05 as before. In general, a confidence region, or a confidence interval for the univariate case, of a level $1 - \alpha$ where $\alpha = 0.05$, defines a 95% interval in which we expect the unknown value of the parameter we try to estimate, in this case the NIS, to fall within this interval with probability at least 95%. In other words, $\alpha = 0.05$, represents the probability of making a type I error (e.g. a false positive decision), by rejecting the null hypothesis when it is true. A type I error in our case means that we incorrectly declare the Kalman filter as inconsistent, and a level $\alpha = 0.05$ defines an upper bound in the probability of committing such an error. For example, assuming 40 time steps ($K = 40$) and one-dimensional measurement vector, the *two-sided confidence region* for the χ^2_{40} variable is $[\chi^2_{40}(0.025), \chi^2_{40}(0.975)]$, which by looking at a χ^2 table we find that the region is defined as $[24.433, 59.342]$. Then, since we want to test whether the time-average normalised innovation squared statistic falls within the interval we divide the lower ($r1$) and upper ($r2$) endpoints by the number of time steps, in this case $K = 40$, to obtain the following interval: $[0.6108, 1.4835]$. For instance, if $\bar{\epsilon}^2_{y_k} = 0.85$, hypothesis H_0 is not rejected (i.e. H_0 is retained), and if it falls outside of the above region (e.g. $\bar{\epsilon}^2_{y_k} = 2.67$), H_0 is rejected.

The aforementioned statistical test is constructed by specifying a formula of the form $\mathtt{R}^{nis\_avg}_{=?}[\,\mathtt{C}^{\leq maxTime}\,]$. First, a formula *nis* is used which calculates the normalised innovation squared statistic. Then, we define a reward function *nis_avg* which divides the *accumulated* result obtained from the previous formula, *nis*, by the maximum time the model runs, and assigns the result to the states of the model. In each transition the cumulative reward, defined as C, is being updated and when it reaches the terminal state, the time-average normalised innovation squared statistic (*nis_avg*) is obtained. The terminal state of the model is defined by the predicate `t=maxTime`. Note that the time-average normalised squared value is obtained by dividing by `(maxTime - 2)`, rather than by `maxTime`. This is because we do not want to take into account in the computation the values of the initial states of the model (e.g. when $t = 0$ or when $t = 1$). The value of `(maxTime - 2)` essentially defines the degrees of freedom of the χ^2 distribution, which means that when `maxTime=20`, the lower and upper bounds are computed for 18 degrees of freedom.

5.4 Summary of Contributions

In this chapter we presented a framework for the modelling and verification of Kalman filter implementations operating on linear discrete-time stochastic systems. This framework is general enough to handle the creation of various different filter variants. The contributions of this chapter can be summarised as follows:

- Novel techniques to systematically construct a Markov model of the filter's operation using truncation and discretisation of the stochastic noise model.
- The generation of non-trivial reward structures for the discrete-time Markov chain, which are computed using linear algebra computations on the matrices and vectors used in the execution of the Kalman filter implementation.
- A novel approach that combines statistical theory and formal verification to model and analyse the execution of a Kalman filter in order to produce guarantees on its performance for a given set of parameters.

CHAPTER 6

VerFilter: Verification Through the Integration of Formal Modelling with Kalman Filter Execution

In this section, we present our tool, VerFilter, which is the software implementation of the framework defined in Sections 5.1, 5.2 and 5.3. The VerFilter tool is written in the Java programming language in order to be seamlessly integrated with the PRISM libraries, which are written in Java as well. In particular, PRISM version 4.4 was used for the development of the tool. Moreover, in VerFilter, several of the numerical linear algebra computations for implementing Kalman filters are done using the Apache Commons Math library [161], while other parts have been manually implemented. The tool and supporting files for the results in Chapter 7 are available from [25].

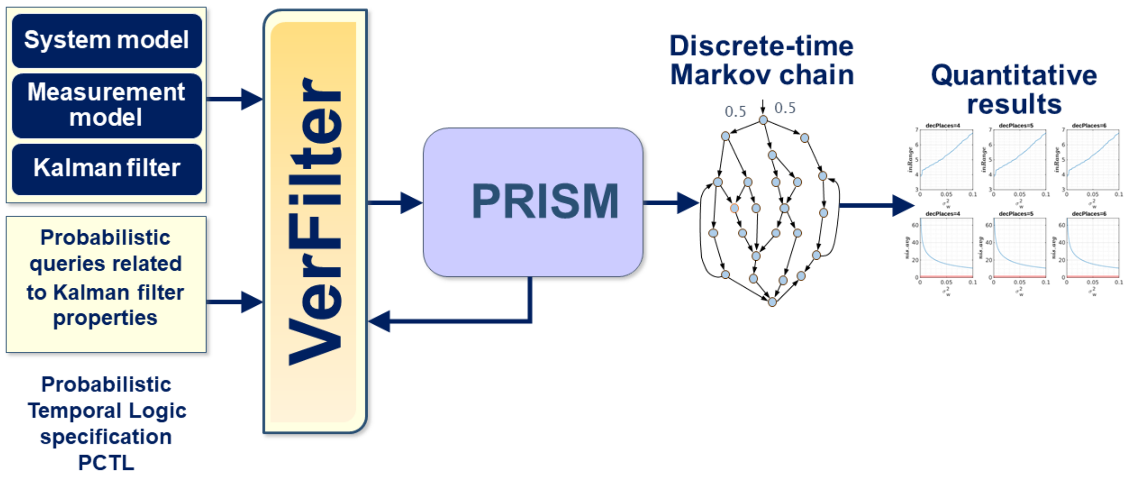

System model
Measurement model
Kalman filter
Probabilistic queries related to Kalman filter properties
Probabilistic Temporal Logic specification PCTL
VerFilter
PRISM
Discrete-time Markov chain
0.5
0.5
Quantitative results

Figure 6.1: A high-level diagram of the approach.

6.1 Overview of VerFilter

Overall approach and workflow. The approach which is supported by VerFilter is the modelling and verification of Kalman filter implementations. In particular, the approach is based on the construction and analysis of a DTMC, representing the behaviour of a particular Kalman filter executing in the context of estimating the state of a linear stochastic dynamical system under a noisy measurement model.

The main three components which are given as inputs to VerFilter, as shown in Figure 6.1, by the user are: (i) the system model; (ii) the measurement model; and (iii) one of the four Kalman filter variants which are described in this chapter. VerFilter takes the mathematical form, as vectors and matrices, of these three components as inputs, similar to what is described in Section 6.2.

The DTMC is automatically constructed based on a specification of the filter and the system whose state it is trying to estimate. This is done through the `gLevel` value which is given as an input by the user and has been explained in detail in Section 5.1. It is also done through the appropriate labelling of states and the computation of their rewards, for the properties. Moreover, the probabilistic model checking queries, which verify properties related to numerical and modelling errors, can also be given as inputs to VerFilter before the model construction process begins.

VerFilter invokes PRISM programmatically and outputs the results of the analysis. In the background, VerFilter executes a stochastic linear dynamical system which emits noisy measurements, and a Kalman filter which tries to estimate the system's state. While this execution is occurring, VerFilter is interacting with PRISM dynamically by updating the DTMC's variables. The interaction with PRISM is done through the `ModelGenerator` interface by overriding its methods, and is presented in detail in Section 6.2.3. More details about the construction of properties and their PCTL expressions can be found in Sections 5.2 and 5.3, for the two classes of properties, respectively.

The precise functions of VerFilter can be summarised as follows:

- Automates the generation of Kalman filters by constructing a DTMC representation of the filter's execution. The execution of the filter and its interaction with PRISM is done automatically through VerFilter. The user needs to specify the inputs shown in Table 6.1 only.
- Returns the verification results to the user directly, without requiring from the user to manually load the model into PRISM.

Assumptions and constraints. While in theory our verification approach would work for verifying higher-dimensional Kalman filters, the current version of VerFilter does not handle Kalman filters of higher dimensionality than two. However, it is worth noting that VerFilter's practical applicability may not be limited as there are many real-world applications which use two-dimensional or even one-dimensional Kalman filters. This is also true of the number of intervals (`gLevel`) into which the noise distribution will be truncated. In theory, the noise distribution can be truncated into more intervals than the six we consider here.

The VerFilter software tool has been made publicly available at [25]. To verify similar types of filters executing to track the states of the system models (e.g. kinematic) we consider here, the user should specify F, P, H, R and the variance of noise, σ_w^2, which is used for the construction of the process noise covariance matrix Q. Also, the user should specify the type of kinematic model and the type of Kalman filter to be executed. The properties which can be used for the verification of Kalman filters, and are available to the user, are the ones we consider in Sections 5.2 and 5.3. However, under different conditions, such as verifying another property on a different filter type, users could potentially modify parts of our code to tailor the verification to their problem domain. The code published online could serve as a guide in this regard.

6.2 VerFilter Inputs

In Table 6.1 we show the user inputs available to VerFilter, by distinguishing which of those refer to the system and measurement models, which refer specifically to the filter models and which are shared between them. The `RealVector` and `RealMatrix` shown in Table 6.1 are implemented as one-dimensional and two-dimensional arrays of type `double`, respectively. VerFilter also takes as inputs four extra parameters: (i) `gLevel` which denotes the granularity level of the noise and has been discussed in Section 5.1; (ii) `decPlaces` which allows the user to input the desired numerical precision in terms of the number of decimal places, to which the numerical values used in the computations will be rounded and stored in the respective Markov chain states. This also determines the numerical precision of the Kalman filter equations which will be propagated forward in time over the possible Markov chain trajectories. This type of flexibility in the representation of numbers is particularly important because it can help the user to construct various models with different degrees of

Table 6.1: User inputs for each of the models.

Input	Description	Used in:	Type
$\hat{x}_0^+$	A posteriori state estimate vector	Filter	`RealVector`
P_0^+	A posteriori estimation-error covariance matrix	Filter	`RealMatrix`
x	State vector	System	`RealVector`
w	Process noise vector	System	`RealVector`
v	Measurement noise vector	System	`RealVector`
F	State transition matrix	Shared	`RealMatrix`
Q	Process noise covariance matrix	Filter	`RealMatrix`
H	Measurement matrix	Shared	`RealMatrix`
R	Measurement noise covariance matrix	Shared	`RealMatrix`
`gLevel`	Granularity of the noise	Shared	`int`
`decPlaces`	Number of decimal places	Shared	`int`
`maxTime`	Maximum time the model will run	Shared	`int`
`filterType`	Type of filter variant	Shared	`int`

precision, and then use VerFilter to verify them. (iii) `maxTime` which determines the maximum time the model will run; and (iv) `filterType` which is the type of filter to be executed.

From a high-level point of view VerFilter comprises four components each of which takes certain inputs in which VerFilter acts upon (see Section 6.3.1 for more details). The first one is the system model which is a linear dynamical system one is interested in modelling. The second component is the measurement model, which can be thought of as the sensor model which captures the measurements emitted from the system model. The third component encompasses the associated noise distributions of the system and the measurement model, respectively. Finally the last component is one of the four Kalman filter variants which is used to estimate the state of the system model in the presence of uncertainty.

6.2.1 VerFilter Arithmetic

In this section, first we explain how VerFilter handles the numerical computations. VerFilter operates on top of PRISM and has to satisfy the constraints set by it. The most important constraint imposed to VerFilter by PRISM is the use of integers for state variables instead of, for example, double precision floating-point numbers.

This means that the numbers have to be rounded to the desired precision given as input and then to be decomposed to their integral and fractional parts, in order to be stored in the Markov chain state, respectively. In addition, since the Kalman filter relies on the updated values to correct its estimates the *integers* stored in Markov chain state are composed back to *doubles* in order to be processed. These procedures take place in several methods in the `Computations` class.

6.2.2 Automating the Generation of Kalman Filters

VerFilter can currently create the following Kalman filter variants: i) the conventional Kalman filter, ii) the steady-state Kalman filter, iii) the Carlson - Schmidt "square-root" filter and iv) the Bierman - Thornton UD filter. This provides an advantage to the user who can choose to experiment with a variety of different filters. In addition, VerFilter is extensible meaning that it can be accessed programmatically through its API from people who wish to add other filters or modify the existing ones. This is done through the `KalmanFilter` interface, which all of our filter variants implement and is shown in Appendix A, and provides a unified way to implement the standard methods in a newly created Kalman filter class.

In the interface we can observe some of the parameters which were discussed previously. For example, the `getStateEstimationVector()` method refers to the state vector x, while the `getErrorCovarianceMatrix()` one, to the estimation-error covariance matrix P. The `predict()` and `correct(RealVector z)` methods refer to the time and measurement updates. Each one of those methods determines whether the state estimate is considered an a priori or an a posteriori estimate. This also applies to the estimation-error covariance matrix P. In the former case x and P are considered a priori while in the latter a posteriori. The inputs with their types for each of the filter classes are shown in Table 6.2.

In the following sections, we will describe how VerFilter handles the creation of various Kalman filter implementations. This will involve switching between different

Table 6.2: User inputs for each of the filters.

Parameter	Description	Type
x_0	State vector	`RealVector`
F	State transition matrix	`RealMatrix`
P_0	Estimation-error covariance matrix	`RealMatrix`
Q	Process noise covariance matrix	`RealMatrix`
H	Measurement matrix	`RealMatrix`
R	Measurement noise covariance matrix	`RealMatrix`

contexts; for example there is the numerical linear algebra context, in which the filter is looked from a mathematical and algorithmic viewpoint.

6.2.3 The ModelGenerator Interface

PRISM's `ModelGenerator` interface, provides us with the necessary method signatures to generate a Markov chain, by overriding them. For easy reference, some of the method signatures inside the `ModelGenerator` interface, are described in Appendix A. The state variables and their associated types are stored in a `List` and are being returned from the `getVarNames` and `getVarTypes` methods, respectively. The `getInitalState` method returns the initial state of the Markov chain, which from an estimation viewpoint can be thought of as the set of the initial conditions. Furthermore, a value of 1 is returned from the `getNumChoices` method, since the only nondeterministic choice is based on the value of the process noise, which perturbs the system model's state vector. The returned values of the `getNumTransitions` and `getTransProbability` methods are determined based on the input value of the `gLevel` variable.

The `computeTransitionTarget` method in the `FilterPrism` class is arguably one of the important methods in VerFilter, since it updates the values of the variables in the states. The variables which define the Markov chain state are shown in Table 6.3. First, we start by fetching the Markov chain state which is explored. The state is returned from the `getExploreState` method, and contains the variables'

Table 6.3: Kalman filter variables which are stored in the Markov chain state

System model variables			
CKF_var	**Type**	**PF_var**	**Type**

`F`	`[][] double`	`F11IntState`
		`F11FracState`
		`F12IntState`
		`F12FracState`
		`F21IntState`
		`F21FracState`
		`F22IntState`
		`F22FracState`
			`Integer`

`xSim`	`[] double`	`x11IntSimState`
		`x11FracSimState`
		`x21IntSimState`
		`x21FracSimState`
			`Integer`

`w`	`double`	`w11IntState` `w11FracState`	`Integer`
Measurement model variables			
CKF_var	**Type**	**PF_var**	**Type**

`H`	`[][] double`	`H11IntState`
		`H11FracState`
		`H12IntState`
		`H12FracState`
			`Integer`

`v`	`double`	`v11IntState` `v11FracState`	`Integer`
`z`	`double`	`z11IntState` `z11FracState`	`Integer`
Kalman filter variables			
CKF_var	**Type**	**PF_var**	**Type**

`xEst`	`[] double`	`x11IntEstState`
		`x11FracEstState`
		`x21IntEstState`
		`x21FracEstState`
			`Integer`

`P`	`[][] double`	`P11IntState`
		`P11FracState`
		`P12IntState`
		`P12FracState`
		`P21IntState`
		`P21FracState`
		`P22IntState`
		`P22FracState`
			`Integer`

names in an array of type `Integer`. Then, we initialise the state estimate of the Kalman filter, `xEst`, by invoking the respective elements of the array stored in the `State` class. The type of `xEst` is a `RealVector` which is implemented as an array of type `double`. From an implementation viewpoint, this means that the estimates stored in the Markov chain state have to be converted to their decimal point counterparts, in order to be stored as vector elements in a `RealVector`, so they can be processed from the `ConventionalKalmanFilter` class correctly. This step is done by considering the numerical precision given as input, in the variable `decPlaces`. Then, the same process is performed for the estimation-error covariance matrix `P`.

Furthermore, the system and measurements model's state vectors are set to their Markov chain counterparts, along with their corresponding noise values. Specifically, the new measurement is computed based on the system model's state vector stored in the Markov chain state and then is stored back into it. Note that the computation of the new measurement is of type `RealVector` and as a result has to be converted back to an `Integer` type, in order to be stored in the Markov chain state. This is done using the `Computations` class which performs computations of this type by taking into account the value of `decPlaces` variable, which is given as an input. The a posteriori variables of the Kalman filter are passed as inputs in the `ConventionalKalmanFilter` class and one iteration of the time and measurement updates is executed. Finally, the newly obtained a posteriori variables are stored in the Markov chain state using the previous procedure.

Moreover, the `getNumLabels` method returns the number of the different labels, while the `isLabelTrue` one checks whether a given label is true or false in a state. There are currently nine labels in VerFilter. For example, the label `end` evaluates to true when `t=maxTime` to check the maximum time the model will run or in other words whether the Markov chain has reached an absorbing state. Finally, the `getStateReward` method returns the reward, a positive real number, of a given

Table 6.4: The conventional Kalman filter algorithm

Time update	**VerFilter's predict() method**
$\hat{x}_k^- = F_{k-1}\hat{x}_{k-1}^+$	`x = F.operate(x);`
$P_k^- = F_{k-1}P_{k-1}^+F_{k-1}^T + Q_{k-1}$	`P = F.multiply(P).multiply(F.transpose()).add(Q);`
Measurement update	**VerFilter's correct(RealVector z) method**
$y_k = z_k - H_k\hat{x}_k^-$	`y = z.subtract(H.operate(x));`
$S_k = H_kP_k^-H_k^T + \mathrm{R}$	`S = H.multiply(P).multiply(H).add(H.getR());`
$K_k = P_k^-H_k^TS_k^{-1}$	`K = P.multiply(H.transpose()).multiply(inverse(S));`
$P_k^+ = (I - K_kH_k)P_k^-$	`P = I.subtract(K.multiply(H)).multiply(P);`
$\hat{x}_k^+ = \hat{x}_k^- + K_ky_k$	`x = x.add(K.operate(y));`

state.

In Section 6.3.1, we show a detailed example of how VerFilter performs an update of the discrete-time Markov chain's state, in the context of particular state estimation problem, by focusing on the `computeTransitionTarget` method.

6.3 The Conventional Kalman Filter

The conventional Kalman filter is the most popular form of the Kalman filter, and it was explained in detail in Section 2.4. This allows us to proceed directly to its VerFilter implementation, as shown in Table 6.4. In the first column we show the mathematical equations of the conventional Kalman filter, and in the second column the VerFilter equivalent written in Java. The state x is the filter's estimate for the system model, while Q is the noise covariance matrix assumed to influence the evolution of the system model from state to state. Usually, in simulations the time and measurement updates operate inside a loop along with the system model. The code which corresponds to the time update is placed directly after the loop starts, then the code for the system is called in order to simulate that the true state of the system model has been perturbed by noise, followed by the measurement model which is also perturbed by random noise. Once the noise measurement has been obtained, the measurement update methods are called where the Kalman filter adjusts its estimate. However, this straightforward structure does not hold in our case.

6.3.1 Conventional Kalman Filter Example

This example presents a situation where the kinematic system model, as defined in Section 2.2.1, is currently 5 metres away from the origin, moving with a velocity of 5 metres per second which is perturbed by noise having a normal distribution with $\mu = 0$ and $\sigma^2 = 0.025$. The sensor, which is assumed to be more accurate, is also perturbed by normally distributed noise with $\mu = 0$ and $\sigma^2 = 0.01$. The task of the Kalman filter is to estimate the system model's state vector, both position and velocity. Note that the velocity can be estimated from the Kalman filter even though it is not directly observed (i.e. it can be inferred from the observable variable, the position). The inputs to VerFilter are the following:

1. **The system model inputs:** $F = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, $x = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$, $\sigma_w^2 = 0.025$
2. **The measurement model inputs:** $H = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $R = 0.01$
3. **The Kalman filter inputs:** $\hat{x}_0^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $P_0^+ = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$,

   $Q = \begin{bmatrix} 0.00833 & 0.0125 \\ 0.0125 & 0.025 \end{bmatrix}$
4. **Numerical precision and granularity level:** `decPlaces=3`, `gLevel=2`

Note that if we swap the place of the 1 and 0 in H, the problem could be framed as trying to estimate the position of the vehicle given the observable velocity values. In either case, this is a problem which could be met in realistic settings. For example, someone drives a car and suddenly in the navigation map which shows the car's position, the GPS signal is lost. Then, software operating in the car could use the Kalman filter to help it localise itself, by determining its position by observing

Table 6.5: The Markov chain state array.
(k is part of the state, but does not appear in the table for space reasons.)

State array																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`z`	`z`	`P`	`P`	`P`	`P`	`P`	`P`	`P`	`P`	`xE`	`xE`	`xE`	`xE`	`xS`	`xS`	`xS`	`xS`

Table 6.6: The Markov chain state array at time $k = 0$.

State array																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	10	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0

the velocity only. The elements of the initial Kalman filter's state vector are set to 0, to denote our complete ignorance with respect to the car's position or velocity. The same applies to the initialisation of P, where the uncertainty of estimate $\hat{x}$ is specified by setting the value of 10 for the position and the velocity. Also, the covariance between these two variables is set to 0, to demonstrate the fact that at time $k = 0$ there is no indication that they vary in a similar way.

The position of the state variables, stored in an array of type `int`, which are going to represent the Markov chain state is shown in Table 6.5. For easier visualisation we have coloured the indices which correspond to: the measurement model with orange, the Kalman filter with blue and the system model with purple. Note that F, R, Q are not stored in the state since time invariance is assumed. The evolution of the Markov chain states, which correspond to one iteration of the VerFilter for the different probability values can be observed in Tables 6.6, 6.7 and 6.8. At time $k = 0$ only the initial values of the Kalman filter variables are stored in the state array; for example the value of 10 in index 2 is the integral part of the first element of the a posteriori covariance matrix P, while index 3 is the fractional part of the same variable.

From the state at the discrete time step $k = 0$, the Markov chain transitions with probability 0.5 to either a state where the process noise w is positive or negative. The mean values for the process and measurement noises are ± 0.126 and ± 0.08, according to the procedure shown in Section 5.1.

Table 6.7: The Markov chain state array at time $k = 1$ reached with probability p.

State array																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	10	0	0	0	10	0	0	0	0	0	0	0	5	0	5	0

At time step $k = 1$, the values of the system model are being set up and the integral parts of the elements of the system model's state vector, indices 14 and 16 are changed to 5 and 5, respectively. From this state if we choose the transition with the positive noise values, the Markov chain transitions to a state at time instant $k = 2$. Let us consider at this point what happens in the `computeTransitionTarget` method. The a posteriori variables are fetched from the Markov chain state and are stored in a vector. This procedure is shown in Listing 6.1.

```
//A posteriori state estimate
x=new ArrayRealVector(new double[]{((int)state.varValues[10]+
((int)state.varValues[11]/pow(10,decPlaces)))),((int)state.varValues[12]+
((int)state.varValues[13]/pow(10,decPlaces)))});
//A posteriori estimation−error covariance
P = new Array2DRowRealMatrix(new double[][]{{(int)explore.varValues[2]+
((int)explore.varValues[3]/pow(10,decPlaces)),(int)explore.varValues[4]+
((int)explore.varValues[5]/pow(10,decPlaces))},{(int)explore.varValues[6]+
((int)explore.varValues[7]/pow(10,decPlaces)),(int)explore.varValues[8]+
((int)explore.varValues[9]/pow(10,decPlaces))}});
```

Listing 6.1: Store the "old" posterior estimates.

Since the initial a posteriori state estimate is zero, the *new* `x` becomes a vector of zeroes. The same procedure is followed for the a priori estimation-error covariance matrix `PEst`. Note also that the `state.varValues[index]` corresponds to the index of the Markov chain state array and that the numerical precision given as input (e.g. `decPlaces`) is taken into account in the computation above.

Next, the a priori state estimate $\hat{x}$ and estimation-error covariance P of the

Kalman filter are computed, and these variables are not stored in the Markov chain's state. It is worth noting that unlike the state estimate $\hat{x}$ in which the order of its execution matters, because it has to be updated *before* the measurement update equation, the same does not apply to the computation of the error covariance equations, since those do not depend on the measurements. These a priori variables will be used to update their a posteriori counterparts.

In Listing 6.2 we show how the a priori computations are implemented in VerFilter.

```
@Override
public void predict(...){
//Computes a priori state estimate
x = F.operate(x);
//Computes a priori error covariance
P=F.multiply(P).multiply(F.transpose()).add(Q);
}
```

Listing 6.2: Computing the time update.

The code above equates to:

$$\hat{x}_2^- = F\hat{x}_1^+ = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{aligned} P_2^- = FP_1^+F^T + Q &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 0.0083 & 0.0125 \\ 0.0125 & 0.025 \end{bmatrix} \\ &= \begin{bmatrix} 20.0083 & 10.0125 \\ 10.0125 & 10.0250 \end{bmatrix} \end{aligned} \tag{6.1}$$

The number of decimal places of the first element in the P_k matrix is explained by the fact that we do not store this matrix in the Markov chain state, and as a result its numerical precision is not limited at this point. Then, the system model variables are fetched from the previous Markov chain state. The variables which correspond to the system model's state vector are stored in a vector with the name `xEntries`. The process noise `w` is stored as a scalar. The update of the system model's state vector `xSim` is implemented as follows:

```
xSim = F.operate(xEntries).mapAdd(w);
```

Listing 6.3: Updating the system model's state vector

The code above is the equivalent of:

$$xSim_2 = FxSim_1 + w \tag{6.2}$$

$$xSim_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 5 \\ 5 \end{bmatrix} + 0.126 = \begin{bmatrix} 10.126 \\ 5.126 \end{bmatrix} \tag{6.3}$$

Unlike the previous case, the updated values of the system's state vector are stored in the Markov chain state, so they have to be converted to integers. To do that, we make use of the methods inside the `Computations` class, as shown in Listing 6.4.

```
int xSim11Int=Computations.getIntegralPartAsInt(xSim.getEntry(0),decPlaces);
int xSim11Fr=Computations.getFractionalPartAsInt(xSim.getEntry(0),decPlaces);
int xSim21Int=Computations.getIntegralPartAsInt(xSim.getEntry(1),decPlaces);
int xSim21Fr=Computations.getFractionalPartAsInt(xSim.getEntry(1),decPlaces);
```

Listing 6.4: Computing the integral and fractional part of a number

This results in 10 and 126, for the integral and fractional parts of the first element of the system's state vector, respectively. The numbers for the second element are 5 and 126. These numbers are then stored in the Markov chain state array in the following indices: 14, 15, 16 and 17. The Markov chain state array is shown in Table 6.8.

The measurement is computed in a similar way; first the corresponding element of the Markov state array is fetched, and then we overwrite it based on the values in the updated system model's state vector. The results of this computation are then stored to positions 0 and 1. For this example, they amount to 10 for the integral and to 206, for the fractional parts, respectively. At this point, the innovation y_k and the innovation covariance S_k, are computed. These variables will be needed for the computation of the Kalman gain K_k. These computations are done in the `correct()` method of the `ConventionalKalmanFilter` class, as shown in Listing 6.5.

```
y = z.subtract(H.operate(x));
S = H.multiply(P).multiply(H.transpose()).add(R);
K= P.multiply(H.transpose()).multiply(inverse(S));
```

Listing 6.5: Computing the innovation, its covariance and Kalman gain.

The equivalent of the code above is:

$$
\begin{aligned}
y_2 &= z_2 - H\hat{x}_2^- \\
&= \begin{bmatrix} 10.206 \end{bmatrix} - \begin{bmatrix} 1 & 0 \end{bmatrix} \bullet \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 10.206 \end{bmatrix} \\
S_2 &= HP_2^-H^T + R \\
&= \begin{bmatrix} 1 & 0 \end{bmatrix} \bullet \begin{bmatrix} 20.0083 & 10.0125 \\ 10.0125 & 10.0250 \end{bmatrix} \bullet \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.01 \end{bmatrix} = \begin{bmatrix} 20.0183 \end{bmatrix} \\
K_2 &= P_2^-H^TS_2^{-1} \\
&= \begin{bmatrix} 20.0083 & 10.0125 \\ 10.0125 & 10.0250 \end{bmatrix} \bullet \begin{bmatrix} 1 \\ 0 \end{bmatrix} \bullet \begin{bmatrix} 0.0500 \end{bmatrix} = \begin{bmatrix} 0.9995 \\ 0.5002 \end{bmatrix}
\end{aligned} \tag{6.4}
$$

The innovation and measurement values are equal at time $k = 2$ (e.g. 10.206), because the filter's a priori state estimate $\hat{x}_k^-$ is zero. Furthermore, in the S_k computation the a priori covariance matrix is used (e.g. P_k^-), which is the one computed in Listing 6.1. Also, note the first element of the Kalman gain matrix, which is very close to 1, which means that the filter essentially neglects the a priori state estimate (e.g. zero state vector in our case), and trusts the measurement. This is because of the relative large values of the a priori estimation-error covariance matrix P_k^-. Now, we have all the information necessary to proceed to the computation of the a posteriori state estimate and its associated covariance matrix, respectively. This is shown in Listing 6.6.

```
//The a priori state estimate is overridden with the a posteriori state estimate
x=x.add(K.operate(y));
//The a priori error covariance is overridden with the a posteriori error covariance
P=I.subtract(K.multiply(H)).multiply(P);
```

Listing 6.6: Computing the a posteriori variables

The code above is equivalent to:

$$\begin{aligned}
\hat{x}_2^+ &= \hat{x}_2^- + K_2 y_2 \\
&= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.9995 \\ 0.5002 \end{bmatrix} \bullet \begin{bmatrix} 10.206 \end{bmatrix} = \begin{bmatrix} 10.2009 \\ 5.1047 \end{bmatrix} \\
P_2^+ &= (I - K_2 H) P_2^- \\
&= \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0.9995 \\ 0.5002 \end{bmatrix} \bullet \begin{bmatrix} 1 & 0 \end{bmatrix} \right) \bullet \begin{bmatrix} 20.0083 & 10.0125 \\ 10.0125 & 10.0250 \end{bmatrix} \\
&= \begin{bmatrix} 0.0100 & 0.0050 \\ 0.0050 & 5.0171 \end{bmatrix}
\end{aligned} \tag{6.5}$$

These are the a posteriori estimates which will be stored in the Markov chain state, after having been rounded to three decimal places. We would like to highlight again the fact that the Kalman filter *inferred* the non-measured velocity value relatively accurate. These values and their indices are passed as inputs to the PRISM's

Table 6.8: The Markov chain state array at time $k = 2$ reached with probability p.

State array																	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
10	206	0	10	0	5	0	10	5	17	10	201	5	105	10	126	5	126

method `setValue(int index, int value)` in order to be stored in the Markov chain state array, after invoking the respective methods in the `Computations` class. The resulting array is shown in Table 6.8.

6.4 The Steady-State Kalman Filter

In this section, we explain how VerFilter handles the creation of the steady-state Kalman filter. It should be pointed out that the fundamental creation mechanism remains similar to the conventional Kalman filter. For example, the user chooses its desired filter variant, and through the interface an instance of the `SteadyStateFilter` class is created. As a clarification, the term "steady-state" refers to the Kalman gain that is in steady state [15]. It turns out that in order to compute the steady-state Kalman gain, the steady-state solution of the a priori error covariance must be found, since the Kalman gain depends on it. Recall from Section 2.6 that this limiting solution can be expressed as the discrete-time algebraic Riccati equation, and the first step before any attempt is made to solve it, is to verify that the necessary conditions hold.

In particular, in Section 2.3, we explained the observability and controllability tests which provide a way of identifying whether a solution exists or not, and for easy reference these tests are restated here. The first theorem states that the matrix pair F, H must be completely observable, which means that the observability matrix M must be of full rank (e.g. $\rho(M) = n$). In VerFilter this is taken care by the `Computations` class in the `getObsMatrix()` method, by first specifying the dimensions of the observability matrix $np \times n$ (np rows and n columns), corresponding to an $n \times n$ matrix F and an $p \times n$ matrix H. Then, we store the elements in each position by iterating over the rows of F, and raising the matrix product (HF) to the corresponding power. The resulting observability matrix M is passed as an argument to the `isObservable` method which returns true or false depending on whether it is singular or not. This step is performed by computing the singular value decomposition on M. Alternatively, it could have been implemented in a slightly different way, by still using a singular value decomposition to find the rank of the matrix M and then subtracting from it the row/column dimension of F. This would have given the number of unobservable states. In the case of zero, it would have meant that none of the states are unobservable.

Once the observability test passes, the controllability test takes place, which seeks to evaluate whether the matrix pair F, C is completely controllable. The procedure followed is relatively similar to the previous case and is implemented in the `getContMatrix()` and `isControllable` methods. The only difference is the extra computation needed for the C matrix, which is a Cholesky factor of the process noise covariance matrix Q. This is handled in the `getCholeskyFactor()` method which returns the triangular factor C which is then passed as an argument in the `getContMatrix()` method along with F. The Cholesky decomposition is computed using the `CholeskyDecomposition` class from Commons Math, which takes as input a positive definite matrix, P in our case, and calculates its Cholesky decomposition CC^T. The fact that this method throws a `NonPositiveDefiniteMatrixException` when the matrix is not positive definite, can be used as an extra verification step which ensures the positive definiteness of the matrix P. This is because the Cholesky decomposition of a symmetric positive definite matrix is *unique*. In summary, if both of the `isObservable` and `isControllable` methods return true, then we can proceed to solving the Riccati equation since it will converge to a finite steady-state covariance which is a unique and positive definite matrix [23].

Pre-computing the steady-state gains has the advantage of reducing the Markov

Table 6.9: The Markov chain state array of the steady-state filter.

State array									
0	1	2	3	4	5	6	7	8	9
`z`	`z`	`xE`	`xE`	`xE`	`xE`	`xS`	`xS`	`xS`	`xS`

chain state space by a factor of ≈ 2. This is because there is no need in storing the a posteriori estimation-error covariance matrix P in the Markov chain state, since it will remain constant throughout the transitions. The reduced Markov chain state array is shown in Table 6.9. On the other hand using fixed Kalman gains implies transforming the Kalman filter from a *time-varying* filter to a time-invariant one. This transformation could hinder the accuracy of the estimates produced by the steady-state filter, because of the suboptimal gains being used in each of the Markov chain states.

6.5 The Carlson-Schmidt Square-Root Filter

In the `Computations` class, the `getUTCholeskyFactor(RealMatrix P)` method accepts as an argument the *reconstructed* estimation-error covariance matrix P and returns an upper triangular Cholesky factor C such that $P = CC^T$, by iterating over P backwards [13].

Schmidt's algorithm for the time update operates on the following partitioned matrix:

$$A = \begin{bmatrix} FC_{P+} & \Gamma C_Q \end{bmatrix} \tag{6.6}$$

where C_Q is a Cholesky factor of the process noise covariance matrix Q, which in our case is just the square root of a scalar, and C_{P+} is a Cholesky factor of the a posteriori estimation-error covariance matrix P^+. The products of the matrices above define the input to Schmidt's time update and triangularisation is performed using Householder transformations, which is a numerically stable method for performing a decomposition of a matrix into the product of an orthogonal and an upper triangular matrix, respectively [162].

Example 6.1. This example shows how VerFilter handles the creation of the Carlson-Schmidt square-root filter to estimate the state of a discrete white noise acceleration model. It is implemented in the `CarlsonSchmidtFilter` class, and the inputs to VerFilter are the following:

1. **The system model inputs:** $\Delta t = 1$, $F = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, $x = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$, $\Gamma = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}$, $\sigma_w^2 = 4$
2. **The measurement model inputs:** $H = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $R = 0.01$
3. **The Carlson-Schmidt filter inputs:** $\hat{x}_0^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $P_0^+ = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$, $Q = \Gamma\sigma_w^2\Gamma^T = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$
4. **Numerical precision and granularity level:** `decPlaces=3`, `gLevel=2`

Note that there is no requirement from the user of VerFilter to perform a decomposition of P, rather those decompositions are being handled internally by VerFilter. Once the inputs have been received, the first step is to compute the upper triangular Cholesky factors of P and σ_w^2, a square root, in order to compute the products that make up the partitioned matrix A. The result of those computations, performed in the `Computations` class, gives the following:

$$FC_{P+} = \begin{bmatrix} 3.1623 & 3.1623 \\ 0 & 3.1623 \end{bmatrix}, \Gamma C_Q = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{6.7}$$

As pointed out by Carlson [56], despite the fact that F and C_{P+} are already upper triangular and as a result their product will result in an upper triangular matrix, we still need a (re)triangularisation method such as the Householder transformation or the matrix root-sum square (Carlson's choice), since the incorporation of the process noise covariance Q in the time update equation causes the triangular form of A to be lost. The output of the `predict` method in the `CarlsonSchmidtFilter` class yields the following upper triangular Cholesky factor C_{P^-}:

$$C_{P^-} = \begin{bmatrix} 3.2733 & 3.2071 \\ 0 & 3.7417 \end{bmatrix} \tag{6.8}$$

such that:

$$P^- = C_{P^-} C_{P^-}^T = \begin{bmatrix} 3.2733 & 3.2071 \\ 0 & 3.7417 \end{bmatrix} \bullet \begin{bmatrix} 3.2733 & 0 \\ 3.2071 & 3.7417 \end{bmatrix} = \begin{bmatrix} 21 & 12 \\ 12 & 14 \end{bmatrix} \tag{6.9}$$

The time update of the a priori state estimate $\hat{x}^-$ remains the same as in the conventional Kalman filter implementations (e.g. $\hat{x}^- = [0\ 0]^T$).

Carlson's part of the algorithm, the "actual" square-root filter, is implemented in the `correct` method in the `CarlsonSchmidtFilter` class. For the measurement update of the square-root of the a posteriori covariance matrix C_{P^+} and the a posteriori state estimate $\hat{x}^+$ Carlson's algorithms works as follows [13, 56, 57]:

1. Initialise $a_0 = R$, $e_0 = 0$, $f = C_{P^-}^T H^T$
2. `for i=1,2...n` loop through lines $3 - 7$.
3. $a_i = a_{i-1} + f_i^2$
4. $b_i = \sqrt{(a_{i-1}/a_i)}$
5. $c_i = f_i / \sqrt{(a_{i-1} a_i)}$
6. $e_i = e_{i-1} + C_{P_i^-} f_i$
7. $C_{P_i^+} = C_{P_i^-} b_k - e_{i-1} c_i$

When the algorithm terminates, C_{P^+} is an upper triangular Cholesky factor of the a posteriori covariance matrix P^+ such that $P^+ = C_{P^+}C_{P^+}^T$. The a posteriori state estimate is given as $\hat{x}^+ = \hat{x}^- + e_n[z - H\hat{x}^-]/a_n$. As an example, consider the C_{P^-} and $\hat{x}^-$ calculated in the time update. Also, consider the R and H matrices as given in Example 6.1. Carlson's algorithm is executed as follows:

1. $a_0 = 0.01$, $e_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $f = \begin{bmatrix} 3.2733 & 0 \\ 3.2071 & 3.7417 \end{bmatrix} \bullet \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3.2733 \\ 3.2071 \end{bmatrix}$
2. `for i=1`:
3. $a_1 = 0.01 + 10.7145 = 10.7245$
4. $b_1 = \sqrt{0.01/10.7245} = 0.0305$
5. $c_1 = 3.2733/\sqrt{0.01 \times 10.7245} = 9.9953$
6. $e_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 3.2733 \\ 0 \end{bmatrix} 3.2733 = \begin{bmatrix} 10.7145 \\ 0 \end{bmatrix}$
7. $C_{P_1^+} = \begin{bmatrix} 3.2733 \\ 0 \end{bmatrix} 0.0305 - \begin{bmatrix} 0 \\ 0 \end{bmatrix} 9.9953 = \begin{bmatrix} 0.1 \\ 0 \end{bmatrix}$
8. `for i=2`:
9. $a_2 = 10.7245 + 10.2855 = 21.01$
10. $b_2 = \sqrt{10.7245/21.01} = 0.7145$
11. $c_2 = 3.2071/\sqrt{10.7245 \times 21.01} = 0.2137$
12. $e_2 = \begin{bmatrix} 10.7145 \\ 0 \end{bmatrix} + \begin{bmatrix} 3.2071 \\ 3.7417 \end{bmatrix} 3.2071 = \begin{bmatrix} 21 \\ 12 \end{bmatrix}$
13. $C_{P_2^+} = \begin{bmatrix} 3.2071 \\ 3.7417 \end{bmatrix} 0.7145 - \begin{bmatrix} 10.7145 \\ 0 \end{bmatrix} 0.2137 = \begin{bmatrix} 0.0018 \\ 2.6734 \end{bmatrix}$

At this point the algorithm terminates and the a posteriori upper triangular Cholesky factor C_{P^+} of the a posteriori covariance matrix P^+ is given as:

$$C_{P^+} = \begin{bmatrix} 0.1000 & 0.0018 \\ 0 & 2.6734 \end{bmatrix} \tag{6.10}$$

and the a posteriori state estimate $\hat{x}^+$ is:

$$\hat{x}^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 21 \\ 12 \end{bmatrix} \left(\left(11.378 - \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) / 21.01 \right) = \begin{bmatrix} 11.3726 \\ 6.4986 \end{bmatrix} \tag{6.11}$$

6.6 The Bierman-Thornton U-D Filter

In order to construct the U-D filter, first, the following two matrices are defined:

$$W = \begin{bmatrix} FU_{P^+} & G \end{bmatrix} \quad D_W = \begin{bmatrix} D_{P^+} & 0 \\ 0 & D_Q \end{bmatrix} \tag{6.12}$$

where the matrix G can be set equal to the noise distribution matrix Γ (as long as rows $\leq$ cols), or can be obtained from the modified Cholesky decomposition [16]. Also, for discretised models wthout an explicit noise distribution matrix Γ, the G matrix can be replaced with the identity matrix I after diagonalising Q. Furthermore, the noise covariance matrix Q is assumed to be a diagonal matrix, represented by D_Q. Then, by applying the Modified Weighted Gram-Schmidt (MWGS) orthogonalisation algorithm [57], the goal is to find U, D factors such that:

$$U_{P^+} D_{P^+} U_{P^+}^T = W D_W W^T \tag{6.13}$$

Example 6.2. This example shows how VerFilter handles the creation of the U-D filter to estimate the state of a discrete white noise acceleration model. It is implemented in the `UDFilter` class, and the inputs to VerFilter are the following:

1. **The system model inputs:** $\Delta t = 1$, $F = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, $x = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$, $\Gamma = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}$, $\sigma_w^2 = 4$
2. **The measurement model inputs:** $H = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $R = 0.01$
3. **The U-D filter inputs:** $\hat{x}_0^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $P_0^+ = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$, $Q = \Gamma \sigma_w^2 \Gamma^T = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$
4. **Numerical precision and granularity level:** `decPlaces=3`, `gLevel=2`

Here we restate the fact that there is no requirement imposed to the user of VerFilter to perform a decomposition of P_0^+ after having chosen the appropriate filter, U-D filter in this case, rather those decompositions are being handled internally by VerFilter. Once the inputs have been entered, the first step is to compute the $U_{P^+} D_{P^+} U_{P^+}^T$ decomposition of P^+, since the $U_{P^+} D_{P^+}$ factors are to be passed in the `predict` method (Thornton's algorithm) of the `UDFilter` class. The decomposition of P_0^+ to a unit upper triangular factor U_{P^+} and to a diagonal factor D_{P^+} which is performed in the `Computations` class, results in the following:

$$P^+ = U_{P^+} D_{P^+} U_{P^+} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.14}$$

The U_{P^+} and D_{P^+} factors to be processed by Thornton's algorithm are therefore:

$$U_{P^+} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} D_{P^+} = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \tag{6.15}$$

The estimation process starts by calling the usual `predict` method which constructs the aforementioned W and D_W matrices. In our case since the variance of noise is treated as a scalar, the W and D_W matrices are constructed as follows:

$$W = \begin{bmatrix} 1 & 1 & 0.5 \\ 0 & 1 & 1 \end{bmatrix} D_W = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 4 \end{bmatrix} \tag{6.16}$$

The output of the `predict` method yields the following U_{P^-}, D_{P^-} factors:

$$U_{P^-} = \begin{bmatrix} 1 & 0.85714 \\ 0 & 1 \end{bmatrix} D_{P^-} = \begin{bmatrix} 10.7143 & 0 \\ 0 & 14 \end{bmatrix} \tag{6.17}$$

such that:

$$P^- = U_{P^-} D_{P^-} U_{P^-}^T = \begin{bmatrix} 1 & 0.85714 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 10.7143 & 0 \\ 0 & 14 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0.85714 & 1 \end{bmatrix} = \begin{bmatrix} 21 & 12 \\ 12 & 14 \end{bmatrix} \tag{6.18}$$

The time update of the a priori state estimate $\hat{x}^-$ remains the same as in the other implementations (e.g. $\hat{x}^- = [0\ 0]^T$).

Bierman's part of the algorithm, the "actual" square-root filter, is implemented in the `correct` method in the `UDFilter` class. For the measurement update of the U_{P^+}, D_{P^+} factors of the a posteriori covariance matrix and the a posteriori state estimate $\hat{x}^+$ Bierman's algorithms works as follows [13, 56, 57]:

1. Initialise $a_0 = R$, $f = U_{P^-}^T H^T$, $v = D_{P^-} f$

2. `for i=1,2...n` loop through lines $3 - 9$.

3. $a_i = a_{i-1} + f_i v_i$

4. $D_{ii}^+ = D_{ii}^- a_{i-1}/a_i$

5. $b_i = v_i$

6. $\lambda_i = -f_i / a_{i-1}$

7. `for j=1,2...,(i-1)`

8. $U^+_{ji} = U^-_{ji} + b_j \lambda_i$

9. $b_j = b_j + U^-_{ji} v_i$

When the algorithm terminates, the U_{P^+}, D_{P^+} factors of the a posteriori covariance matrix P^+ have been computed such that $P^+ = U_{P^+} D_{P^+} U^T_{P^+}$. The Kalman gain K is given as $K = b/a_n$ and the a posteriori state estimate is given as $\hat{x}^+ = \hat{x}^- + K[z - H\hat{x}^-]$. As an example, consider the U_{P^-}, D_{P^-} and $\hat{x}^-$ calculated in the time update. Also, consider the R and H matrices as given in Example 6.2. Bierman's algorithm is executed as follows:

1. $a_0 = 0.01$, $f = \begin{bmatrix} 1 & 0 \\ 0.85714 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.85714 \end{bmatrix}$, $v = \begin{bmatrix} 10.7143 \\ 12 \end{bmatrix}$

2. `for i=1:`

3. $a_1 = 0.01 + 1 * 10.7143 = 10.7243$

4. $D_{11} = 10.7143 * 0.01 / 10.7143 = 0.01$

5. $b_1 = 10.7143$

6. $\lambda_1 = -1/0.01 = -100$

7. `for i=2:`

8. $a_2 = 10.7243 + 0.85714 * 12 = 21.01$

9. $D_{22} = 14 * 10.7243 / 21.01 = 7.1461$

10. $b_2 = 12$

11. $\lambda_2 = -0.85714/10.7243 = -0.079925$

12. `for j=1:`

13. $U_{12} = 0.85714 + (10.7143 * -0.079925) = 0.0008$

14. $b_1 = 10.7143 + 0.85714 * 12 = 21$

At this point the algorithm terminates and the a posteriori U, D factors of the a posteriori covariance matrix P^+ are given as:

$$U_{P^+} = \begin{bmatrix} 1 & 0.0008 \\ 0 & 1 \end{bmatrix} \tag{6.19}$$

$$D_{P^+} = \begin{bmatrix} 0.01 & 0 \\ 0 & 7.1461 \end{bmatrix} \tag{6.20}$$

$$\tag{6.21}$$

and the a posteriori state estimate $\hat{x}^+$ is:

$$\hat{x}^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 21 \\ 12 \end{bmatrix} \Big(11.378 - \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Big) = \begin{bmatrix} 11.3726 \\ 6.4986 \end{bmatrix} \tag{6.22}$$

Note that the results above agree with the ones obtained from Carlson's algorithm. Specifically, the a posteriori covariance matrix obtained from Bierman's algorithm (UDU^T) agrees with the one obtained from Carlson's (CC^T). The same also applies to the Kalman gain, and consequently to the a posteriori state estimate.

6.7 Summary of Contributions

The contributions of this chapter can be summarised as follows: The software implementation of the novel techniques which were described in Chapter 5, including a detailed example of its usage and its application to several filter implementations.

CHAPTER 7

Evaluating Kalman Filter Verification

In this chapter we illustrate results from the implementation of our techniques on the filter variants described and implemented in Chapters 5 and 6, respectively. Our experimental results target the two classes of properties which were explained in Sections 5.2 and 5.3. In particular, in Section 7.1 we present results on verifying numerical stability properties of the conventional Kalman filter, the Carlson-Schmidt square-root filter and the Bierman-Thornton U-D filter. This is followed by a scalability analysis in terms of the model construction and model checking times, in Section 7.1.3.

Next, in Section 7.2 we present our rationale for using a kinematic state model as an approximation for modelling the CPU utilisation of VMs operating in the cloud. In Section 7.3 we show results for verifying properties which relate to modelling error compensation techniques. The types of Kalman filters we consider are the conventional Kalman filter and the steady-state Kalman filter. Finally, in Section 7.4 we summarise the main contributions of this chapter. The tool and supporting files for the results are available from [25].

7.1 Verification of Numerical Stability of Kalman Filter Implementations

7.1.1 Verification Methodology

The research question the experiments in this section address is whether formal verification, and probabilistic model checking in particular, can be used to verify properties which relate to numerical errors in Kalman filters. Towards addressing this research question, the methodology used for the design of experiments allows us to investigate first whether the approach presented in Chapter 5 and its implementation in Chapter 6 can be used reliably to formally verify the numerical properties which were presented in Section 5.2.

Moreover, the results presented here have been checked against programming language implementations which are used for numerical analysis such as MATLAB. The associated files which have been made available online [25] allow for the reproducibility of the results of the experiments. Once the necessary validity has been established, we analyse the performance characteristics of our approach in Section 7.1.3, in terms of its scalability, and we proceed to addressing the second part of our main research question in Section 7.3.

For the system models in our experiments, we use two distinct *kinematic state models* which describe the motion of objects as a function of time. For the first, the discrete white noise acceleration model, the initial estimation-error covariance matrix P_0^+ is defined as $\begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$. Defining P_0^+ as a diagonal matrix is quite common, since it is initially unknown whether the state variables are correlated to each other. The process noise covariance matrix is given by $Q = \Gamma\sigma_w^2\Gamma^T$ where the noise gain matrix $\Gamma = [\frac{1}{2}\Delta t^2 \ \Delta t]^T$ is initialised by setting the sampling interval Δt to 1, which results in $\Gamma = [0.5 \ 1]^T$. The variance σ_w^2 is set to 0.001 initially. For the second model, the continuous white noise acceleration model CWNA, σ_w^2 is initially set to

0.001. Note that each of these models results in a different process noise covariance matrix Q.

7.1.2 Experimental Results

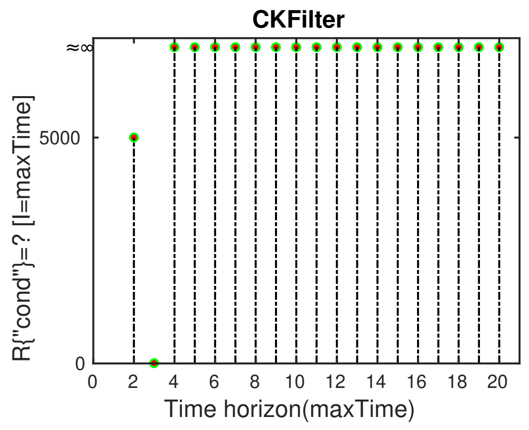
In the first set of experiments, shown in Figure 7.1 we analyse the condition numbers of P^+_{CKF}, P^+_{UDF} for the conventional (`CKFilter`), U-D filter (`UDFilter`), and C^+ for the square-root filter (`SRFilter`), respectively. This is in order to verify that P^+ or the matrices which constitute P^+, remain well-conditioned in terms of maintaining their nonsingularity as they are being propagated forward in time (as discussed in Section 5.2). Note that for the U-D filter we verify the reconstructed matrix P^+ at a specific time step, while for the square-root filter, the Cholesky factor C^+. Then, this property is verified against two inputs which we vary; the first is the numerical precision in terms of the number of decimal places, which we vary from 3 to 6 inclusive. The second input is the time horizon of the model which in our case is measured in discrete time steps and is varied from 2 to 20.

Our goal is twofold. Firstly, we examine whether an increase in the numerical precision has a meaningful effect on how accurately the condition number is computed. This is important since, as we show in Section 7.1.3, a decrease in the numerical precision usually makes verification more efficient. Being able to consider an appropriate threshold above which an increase in the numerical precision is unlikely to have an effect on the property to be verified can determine the applicability of these verification mechanisms in realistic settings. Secondly, we examine whether letting the model evolve for a greater amount of time could have an impact on the property that is being verified. The first observation for the `CKFilter` as shown in Figures 7.1a and 7.1b is that the increased numerical precision actually determines the verification result. For example, we note that for `maxTime` values in the range of $[4 - 20]$, when the input to our model for the numerical precision is 3 decimal places, the instantaneous reward jumps to infinity. An infinite reward in this case means that the condition number of P^+ is $\approx$ 1.009e+16, which practically means that P^+ is "computationally" singular and consequently positive definiteness is not being preserved. This is in contrast to the result obtained for both the `SRFilter` and `UDFilter`, where positive definiteness is preserved, irrespective of the numerical precision and the value of `maxTime` used. For the `SRFilter`, as shown in Figures 7.1c and 7.1d, the highest reward value is ≈ 70, while for the `UDFilter`, Figures 7.1e and 7.1f, it is ≈ 5000. This means that the positive definiteness property of P^+ is maintained throughout the execution of these two types of filters, and the increase in the numerical precision did not have any significant effect on the propagated matrix P^+.

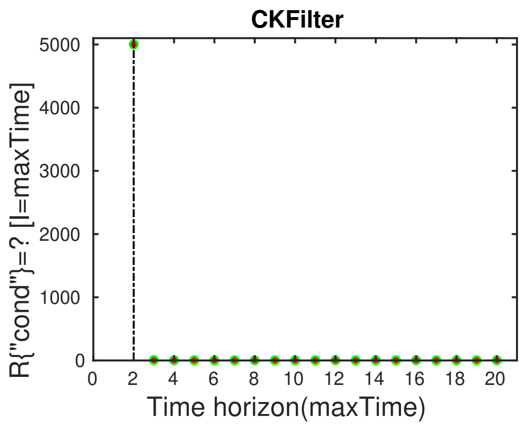
On the other hand, for the `CKFilter`, positive definiteness is preserved when we increase the numerical precision to a value > 4, and the instantaneous reward assigned to the states fluctuates around small values close to zero. Another interesting observation is that for the three filters considered, the instantaneous rewards stabilise to a value of ≈ 2, irrespective of whether the numerical precision is 4, 5 or 6. In fact, for the three filter variants, the actual absolute difference of the rewards over the states in which positive definiteness is preserved between a numerical precision of 5 and 6 decimal places, is ≈ 0.1.

In the second set of experiments, we consider the `CKFilter` and `SRFilter` while the system model is a continuous white noise acceleration kinematic model. Our goal is to examine how VerFilter can be used to examine heuristic-based approaches and ad-hoc methods such as artificial noise injection in terms of their usefulness in correcting potential numerical problems in P^+. This is also helpful in situations where it is challenging to determine the elements of Q, by performing an automatic search over those values which will produce an optimal performance, in this case in terms of the numerical robustness of P.

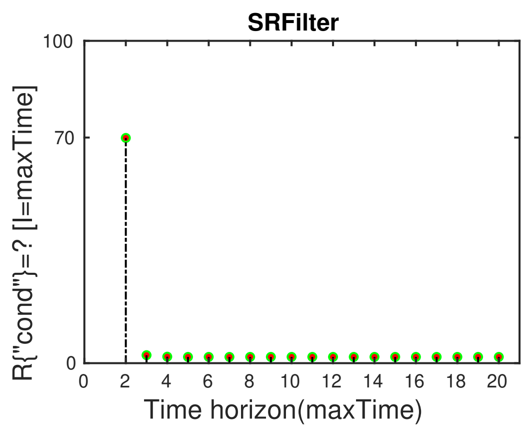
To this end, we verify whether P^+ will remain well-conditioned or not, by varying the elements of Q. Similar to the first set of experiments we verify C^+ directly. For



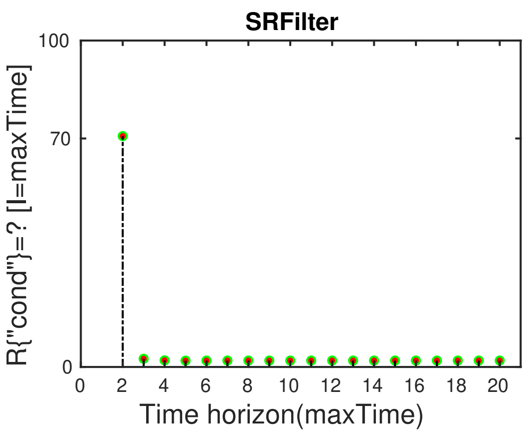
(a) 3 decimal places



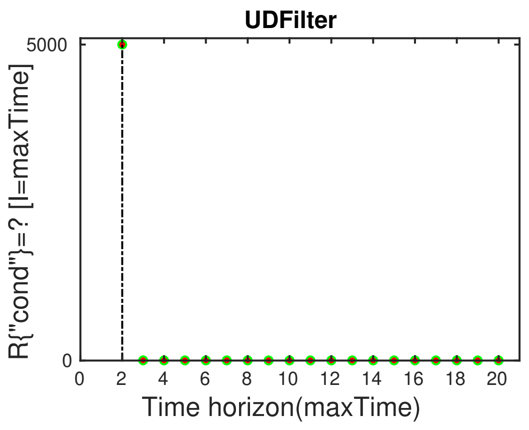
(b) 4 - 6 decimal places



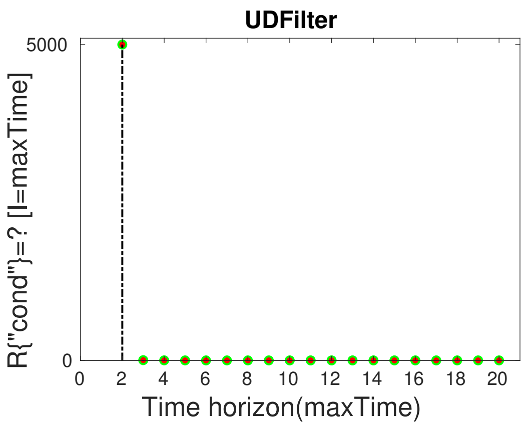
(c) 3 decimal places



(d) 4 - 6 decimal places



(e) 3 decimal places



(f) 4 - 6 decimal places

Figure 7.1: Condition number of $P+$ over time under various degrees of precision.

both of the filters, the noise variance σ_w^2, which determines the elements of Q, is the input to our model, P^+ (and C^+) is being verified against. We do not vary the maximum time; rather, we let the Markov chain to a fixed `maxTime` value of 20 time

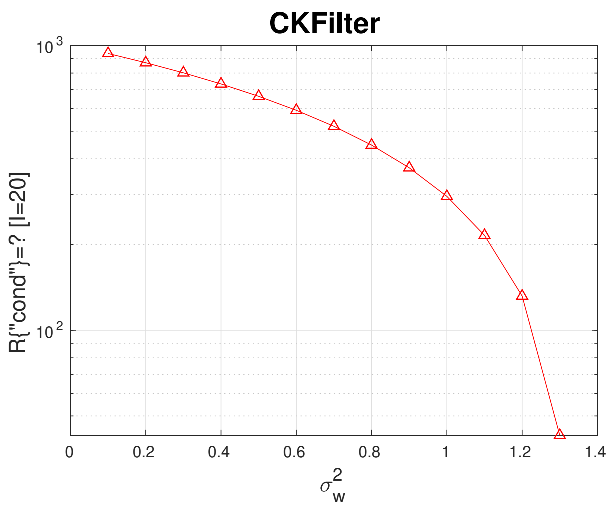


Figure 7.2: Verifying goodness of P^+

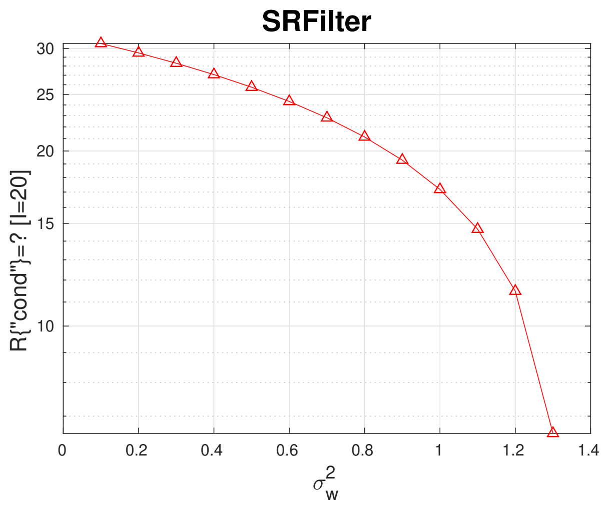


Figure 7.3: Verifying goodness of P^+

steps, which corresponds to $\approx 1 \times 10^6$ states. Also, the numerical precision is fixed to a number of 3 decimal places.

In Figures 7.2 and 7.3 we show the effects of increasing the variance of the noise by small increments, which is then multiplied with the elements of Q, for the `CKFilter` and the `SRFilter`, respectively. The first point of the plot $(0.1, 1000)$ in Figure 7.2, means that for a value of $\sigma_w^2 = 0.1$, the corresponding instantaneous reward which corresponds to the condition number of P^+ in a set of states where `maxTime`=20, is 1000. Similarly, the first point of the plot $(0.1, 30)$ in Figure 7.3, implies that the corresponding instantaneous reward for C^+ for a `maxTime`=20 is

≈ 70. For both of the filter types, a similar trend is observed. As we increase σ_w^2, the reward values decrease which implies an increase on the "quality" of P^+s. In the end, the reward values reach a condition number of ≈ 43 and ≈ 1.94 for the two filters, respectively.

In summary, for this particular case, the optimal $\sigma_w^2 = 1.3$, for both filters. Especially for the `CKFilter`, it is important to note that when performing verification on Markov chains whose trajectories evolve over multiple states, to verify that the positive definiteness of P^+ is not destroyed between successive states (i.e. successive time steps). To this end, it is advisable to use a property of the form $\mathtt{P}_{=?}[\,\Box\, isPD\,]$ and reject models in which the previous property is not satisfied with probability one.

In Table 7.1 we compare three of the Kalman filter variants available in VerFilter; the `CKFilter`, the `SRFilter` and the `UDFilter`. In this set of experiments, the setup is similar to the first one, in terms of the system model and the filter parameters used.

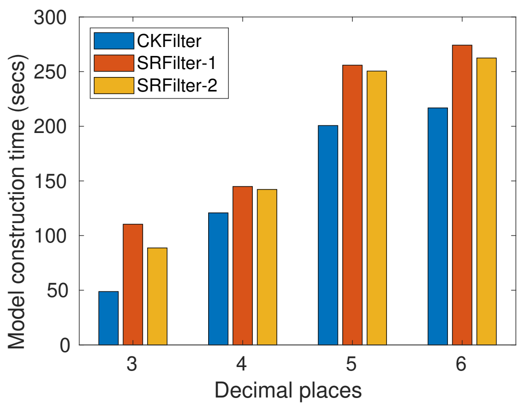
First, our purpose is to demonstrate the correctness of our approach by comparing the condition numbers of P^+_{CKF} and C^+, between the three filter types. For instance, when the positive definiteness property is not violated, and as the reward values begin to converge to certain values, then the difference of the condition number of P^+ between the `CKFilter` and `UDFilter` should be small. The same can also be said for the difference between the condition number of P^+ of the `CKFilter` (and the `UDFilter`), and the squared value of the condition number of C^+ of the `SRFilter`. For example, one can quickly observe in Table 7.1 that the reward values of the `CKFilter` are nearly the same with the respective values of the `UDFilter`. In particular, as the time horizon increases the values of these two filters are becoming equal to each other. Similarly, one can observe that the squared reward values of the `SRFilter` are approximately the same with the reward values obtained for the other two filters. For instance, for a numerical precision of 6 decimal places and a `maxTime` value of 10, the reward value for the `SRFilter` is: $\mathtt{R}_{I=10}^{cond} = 1.966205$. Squaring this result $1.966205^2 \approx 3.866$, which is a very close approximation to the reward values of the other two filters.

The superiority of the `SRFilter` and of the `UDFilter` compared to `CKFilter`, is demonstrated when the numerical precision is 3 decimal places, and mainly from the fact that for the same set of parameters the numerical robustness of P^+ is preserved. This can be seen by comparing the computed results of the reward-based properties as shown in the third,fourth and fifth columns of Table 7.1. We note that, for a numerical precision of 3 decimal places, when choosing the `CKFilter`, the reward value shoots up to $+\infty$, representing a covariance matrix in which the positive definiteness property is destroyed, while in the `SRFilter` and `UDFilter` cases the corresponding reward values settle around the small values of 1.94 and 2.20, respectively. This is also evident by observing the first, second and third columns of Table 7.1 which tell us whether the *isPD* invariant will be maintained in all the states of the model. Notably, the positive definiteness property in the `CKFilter` does not hold for every state, in fact the probability is zero, while for the `SRFilter` the positive definiteness property holds for every state with probability one.

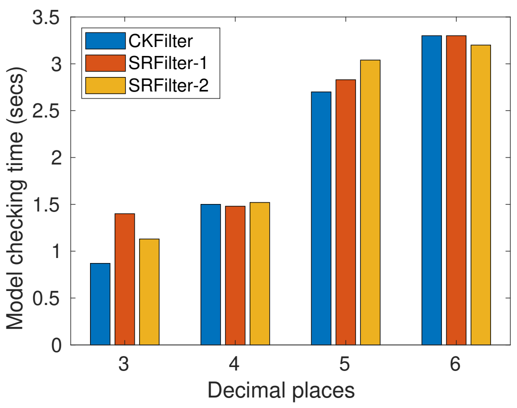
Moreover, when the numerical precision is increased in the range of $[4-6]$ decimal places, the performance of the three filters in terms of the numerical stability of P^+, is roughly the same, and the positive definiteness property of P^+ is maintained in every state considered. One can also note that the reward values of the `CKFilter` and of the `SRFilter` converge to the same value for a numerical precision ≥ 5 decimal places.

Table 7.1: Comparison between three filter variants.

decPlaces	**CKFilter** $\mathtt{P}_{=?}[\,\mathtt{G}\ isPD\,]$	**SRFilter** $\mathtt{P}_{=?}[\,\mathtt{G}\ isPD\,]$	**UDFilter** $\mathtt{P}_{=?}[\,\mathtt{G}\ isPD\,]$	**CKFilter** $\mathtt{R}^{cond}_{=?}[\,\mathtt{I}^{=maxTime}\,]$	**SRFilter** $\mathtt{R}^{cond}_{=?}[\,\mathtt{I}^{=maxTime}\,]$	**UDFilter** $\mathtt{R}^{cond}_{=?}[\,\mathtt{I}^{=maxTime}\,]$
3,maxTIme=2	1	1	1	5001	69.875	5000
3,maxTIme=3	1	1	1	6.854102	2.479506	3.472570
3,maxTIme=4	0	1	1	$+\infty$	2.016643	2.370931
3,maxTIme=5	0	1	1	$+\infty$	1.943289	2.200921
3,maxTIme=6	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=7	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=8	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=9	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=10	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=11	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=12	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=13	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=14	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=15	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=16	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=17	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=18	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=19	0	1	1	$+\infty$	1.943289	2.192302
3,maxTIme=20	0	1	1	$+\infty$	1.943289	2.192302
4,maxTIme=2	1	1	1	5001	70.768988	5000.800100
4,maxTIme=3	1	1	1	6.376508	2.505185	6.784395
4,maxTIme=4	1	1	1	3.439411	2.040158	4.519086
4,maxTIme=5	1	1	1	3.614225	1.939357	3.907120
4,maxTIme=6	1	1	1	3.614225	1.951946	3.866359
4,maxTIme=7	1	1	1	3.614225	1.964276	3.866359
4,maxTIme=8	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=9	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=10	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=11	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=12	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=13	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=14	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=15	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=16	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=17	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=18	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=19	1	1	1	3.614225	1.962769	3.866359
4,maxTIme=20	1	1	1	3.614225	1.962769	3.866359
5,maxTIme=2	1	1	1	5001.060113	70.722645	5000.810100
5,maxTIme=3	1	1	1	6.307259	2.507737	6.322909
5,maxTIme=4	1	1	1	4.164446	2.031345	4.116405
5,maxTIme=5	1	1	1	3.758763	1.932001	3.801081
5,maxTIme=6	1	1	1	3.866359	1.954296	3.852360
5,maxTIme=7	1	1	1	3.866359	1.966188	3.866359
5,maxTIme=8	1	1	1	3.866359	1.966978	3.866359
5,maxTIme=9	1	1	1	3.866359	1.966224	3.866359
5,maxTIme=10	1	1	1	3.866359	1.966224	3.866359
5,maxTIme=11	1	1	1	3.866359	1.966224	3.866359
5,maxTIme=12	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=13	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=14	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=15	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=16	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=17	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=18	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=19	1	1	1	3.866359	1.962769	3.866359
5,maxTIme=20	1	1	1	3.866359	1.962769	3.866359
6,maxTIme=2	1	1	1	5001.062113	70.720408	5000.812100
6,maxTIme=3	1	1	1	6.292646	2.507687	6.283849
6,maxTIme=4	1	1	1	4.129379	2.031178	4.125259
6,maxTIme=5	1	1	1	3.735914	1.931918	3.732897
6,maxTIme=6	1	1	1	3.831732	1.954872	3.820638
6,maxTIme=7	1	1	1	3.874810	1.966389	3.868675
6,maxTIme=8	1	1	1	3.866359	1.966605	3.867762
6,maxTIme=9	1	1	1	3.866359	1.966262	3.866359
6,maxTIme=10	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=11	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=12	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=13	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=14	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=15	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=16	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=17	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=18	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=19	1	1	1	3.866359	1.966205	3.866359
6,maxTIme=20	1	1	1	3.866359	1.966205	3.866359



(a) Model construction time



(b) Model checking time

Figure 7.4: Time comparisons between three filters.

7.1.3 Scalability Analysis

In this section we report on the scalability of our approach in terms of the model construction and model checking time, across three filter variants. The model has been generated by letting the Markov chain evolve to a fixed `maxTime` value of 20 time steps, which corresponds to $\approx 1 \times 10^6$ states. The rationale behind this section is to emphasise the careful analysis that needs to be performed to systematically evaluate the trade-offs between the accuracy of the verification result and the fastness of the verification algorithms.

In Figure 7.4 we show the time comparisons, for varying degrees of precision, between a model which encodes the conventional Kalman filter (`CKFilter`), and our two implementations of the Carlson-Schmidt square-root filter with (`SRFilter-1`) and without (`SRFilter-2`) reconstruction of the covariance matrix, respectively. The model checking time refers to the collective time it takes to compute the first and second property of Section 7.1.2. These sets of experiments were run on a 16GB RAM machine with an i7 processor at 1.80GHz, running Ubuntu 18.04.

By observing Figure 7.4a it is apparent that the increased numerical precision affects the construction time of the models. The average model construction time of the three filter variants increased by a factor of ≈ 3 from 3 to 6 decimal places.

Specifically, the average time is $\approx$ 83 seconds for 3 decimal places compared to $\approx$ 249 seconds, when 6 decimal places were used. Moreover, the construction of the `CKFilter` was the fastest in all the degrees of precision considered, however, as it was noted in Section 7.1.2 it produces an inaccurate verification result when the number of decimal places is 3.

Conversely, the construction times of the two square-root filters were about the same, and it seems that the extra computational step ($P = CC^T$) did not have a significant effect on the performance of the model construction. However, it should be borne in mind that these experiments were conducted on systems represented by two-dimensional matrices. The model checking times are shown in Figure 7.4b and one can observe that they follow a similar pattern with the model construction times shown earlier, in terms of the increase in time from 3 to 6 decimal places. For instance, the average model checking time increases by a factor of ≈ 3 when 6 decimal places are used, compared to 3.

Another observation is that the model checking time appears to be independent of the type of the filter used. This can be seen from the limited variability the model checking time experiences between the three filter variants, since for the degrees of precision considered, it remains at approximately the same level. This is in contrast to the model construction time which appears to be affected by the filter type, since it is considerably less for the `CKFilter` compared to its square root variants. The reason for this seems to be the extra computational steps (e.g. Householder transformations, generation of upper triangular Cholesky factors), which have to be performed in the time update (Schmidt part) and measurement update (Carlson part) of the Carlson-Schmidt square-root filter. In fact, for a precision of 6 decimal places, and once `CKFilter` is chosen as an input we experience a drop in the model construction time of about 53 seconds. However, for the same amount of precision, the time it takes to model check all the three filters is around 3 seconds.

7.2 Cloud System Models

Next, we evaluate Kalman filter implementations on cloud models. First, we explain the rationale behind our modelling approach for specifying a performance model, for the CPU utilisation of VMs running on the cloud. The performance (system) model we apply for the evolution of the CPU utilisation over time relies on kinematic state models. The first reason is that kinematic models are used to model linear stochastic dynamical systems. We argue that the CPU utilisation can be modelled as a linear stochastic dynamical system and in particular by a stochastic linear difference equation of the form $x_{k+1} = x_k + w_k$. This argument is also reinforced from several studies in the literature, for instance the works of [10, 11, 163, 123], who model the CPU utilisation of the *virtualised* components as a one-dimensional random walk. Also, the noise component of the system's state vector, in this case the CPU utilisation, is in accordance with reality since the environment in which the VM runs is inherently stochastic. For example, a potential source of noise includes the variations in the workload between successive time steps, such as requests being added or removed from a server, and it is assumed they follow a Gaussian distribution [11]. Finally, in the measurement equation $z_k = x_k + v_k$, as shown in Figure 7.5, z_k corresponds to the value of CPU utilisation which is actually measured by a computer system monitor tool (e.g. iostat, Amazon CloudWatch, etc.), perturbed by noise, denoted as v_k.

The second reason for choosing a kinematic model is that we believe that notions from physics and classical mechanics in particular, such as position, velocity and acceleration can be applied to model the CPU utilisation over a period of time. For example, in the same way that the velocity of a car is influenced by various disturbances such as wind drag and road bumps, the velocity of CPU utilisation is influenced by the workload changes. A sudden increase in the requests the VM receives means an increase in the rate of change of speed of the CPU utilisation. To reinforce our claim we draw inspiration from the field of quantitative finance, in

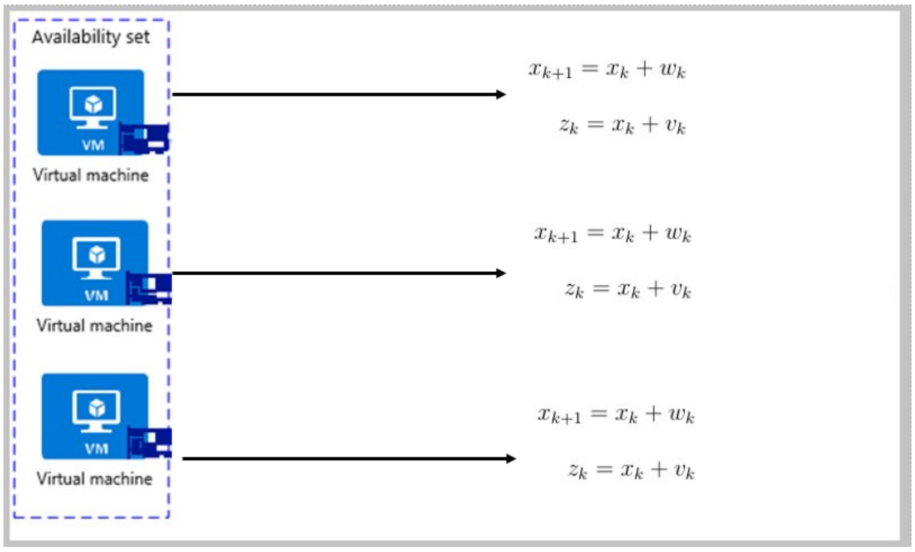


Figure 7.5: A system model of the CPU utilisation for VMs running on the cloud.

which motion models have been applied to model the motion of quantities of interest. For example, notions such as "velocity of a stock price" or "price acceleration" are well-defined in this field, and are being used for prediction and to better understand the market dynamics. One notable example of that is the Black-Scholes model [164], in which it is assumed that the price of a stock follows a geometric Brownian motion. In fact, the influence of physics in economics, in terms of the analogies which can be drawn between physical and financial market phenomena[165], has been such that the interdisciplinary field *Econophysics* has emerged as a result of that [166].

The third reason is efficiency. Unlike some of the previous works (e.g. [163]) in which the state transition matrix F is assumed to have no effect on the system's state vector, in our work we use the state transition matrix F of the kinematic models. In these works, although not cloud-specific, F is simply the identity matrix. For example, for the VMs depicted in Figure 7.5, $F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. One could argue that this is a simplistic model to model the CPU utilisation of VMs in the cloud and could potentially cause performance issues. This is because the number of rows and columns (or the dimension of the matrix) in F scale linearly with the number of

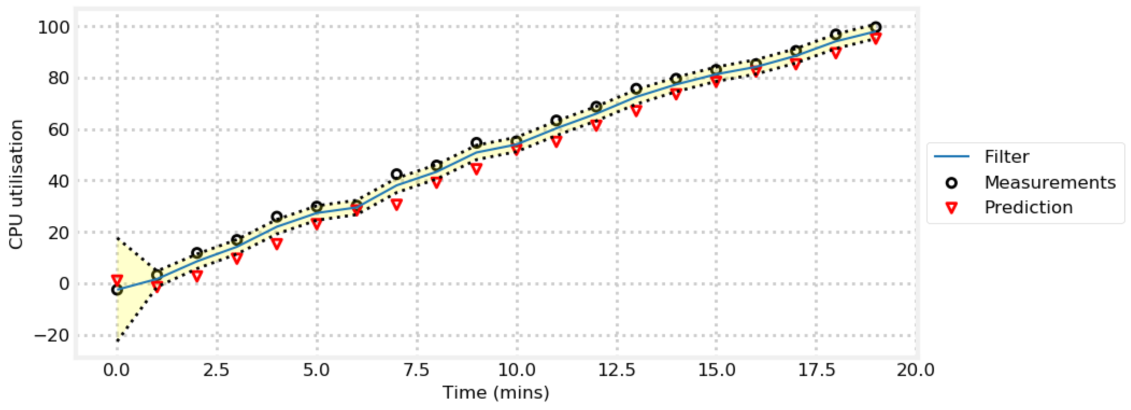


Figure 7.6: A sample CPU utilisation trace.

VMs added in the availability set (Microsoft Azure's terminology) or in the auto-scaling group (Amazon EC2's terminology). As a result, instead of storing in the system's state vector x the CPU utilisation values of all the VMs, we store the average CPU utilisation of all the VMs currently active in the auto-scaling group. This is in accordance to how auto-scaling is performed in a realistic setting, where the auto-scaling decision is taken based on the reported value of an auto-scaling group, which summarises the performance of the individual machines currently in it (e.g. average CPU utilisation).

Example 7.1. The usefulness of a Kalman filter in a cloud-based scenario can be observed in Figure 7.6. The black dots in this figure denote the CPU utilisation measurements which are emitted either from the VMs in real-time or from a simulation in a controlled environment. The red triangles are the predictions of the Kalman filter for the future CPU utilisation values, and the points in the blue line between the prediction and the measurement denote the a posteriori CPU utilisation estimates of the filter. In fact, the a posteriori estimates will *always* lie between the predicted and measured values. Finally, the yellow area represents the variance of the estimation error, which ideally should converge, as in this figure.

7.3 Verification of Kalman Filter Implementation for Cloud System Models

7.3.1 Verification Methodology

Having validated the verification ability of our approach and its software implementation (VerFilter) in Section 7.1.2, we now proceed to addressing the second part of the main research question. Specifically, the research question which is addressed by these experiments is whether we can use formal verification, and probabilistic model checking in particular, to produce performance guarantees for resource estimation mechanisms in cloud computing.

Towards addressing this research question, the methodology used for the design of experiments allows us to investigate whether the approach presented in Chapter 5 and its implementation in Chapter 6 can be used to formally verify properties related to modelling errors in Kalman filters which operate in a cloud computing context, to track CPU utilisation.

In particular, we focus on verifying the so-called consistency of the conventional (`CKFilter`) and of the steady-state Kalman filters, respectively. The consistency criteria of these two different Kalman filter variants, are verified under a wide range of different process noise covariance matrices. These criteria are specified in the form of properties, and have been described in Section 5.3. The performance model of CPU utilisation is defined as a continuous white noise acceleration kinematic model, and its noise covariance matrix is initially defined as $Q = \begin{bmatrix} \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 \\ \frac{1}{2}\Delta t^2 & \Delta t \end{bmatrix} \sigma_w^2$. The sampling interval, Δt, is set to 1, and the measurement noise variance R to 10.

Alternatively, our goal can be described as verifying the process noise covariance matrix tuning, as an effective filter tuning technique for finding "optimal" noise covariance matrices in terms of satisfying the respective consistency criteria. In particular, we analyse by how much the elements of the process noise covariance matrix should be scaled. Before each one of the Kalman filter variants is given as an input to VerFilter, to handle the creation of the probabilistic model, their process noise covariance matrices are multiplied by the variance of noise that perturbs the CPU utilisation (i.e. σ_w^2). Then, we vary σ_w^2 over a range of values, and probabilistic model checking is used to verify certain consistency properties over all possible trajectories of the Markov chain.

In our experiments, we vary σ_w^2 between $[0.001, 5.5]$ inclusive. From 0.001 to 0.1 and from 0.1 to 5.5, the σ_w^2 value is increased in increments of 0.001 and 0.1, respectively. This results in 135 different process noise covariance matrices, against which the conventional and the steady-state Kalman filters are verified. Furthermore, since the impact of the numerical precision, in terms of the number of decimal places, on the verification result is also of interest, the `decPlaces` value is also varied between 3 and 6 inclusive. It is important to state again the two important preconditions of controllability and observability, which have to be satisfied before the verification of the steady-state Kalman filter can begin. The theory underlying these preconditions and our implementation on VerFilter to ensure their satisfaction have been discussed in Section 2.3 and in Section 6.4, respectively.

7.3.2 Results

In this section we present our verification results for the conventional and for the steady-state Kalman filters. In the first set of experiments, we verify the consistency of these two filter types, for noise variance (σ_w^2) values in the range $[0.001, 0.1]$. The first property we verify is whether the magnitude of innovation is bounded by its variance 95% of the time.

In Figures 7.7 and 7.8, we show the expected number of states, for the two filter variants, in which the value of the innovation falls between two standard deviations of the mean (y-axis), for different σ_w^2 values (x-axis). Equivalently, the graphs in Figures 7.7 and 7.8 show the impact of tuning the process noise covariance matrix on

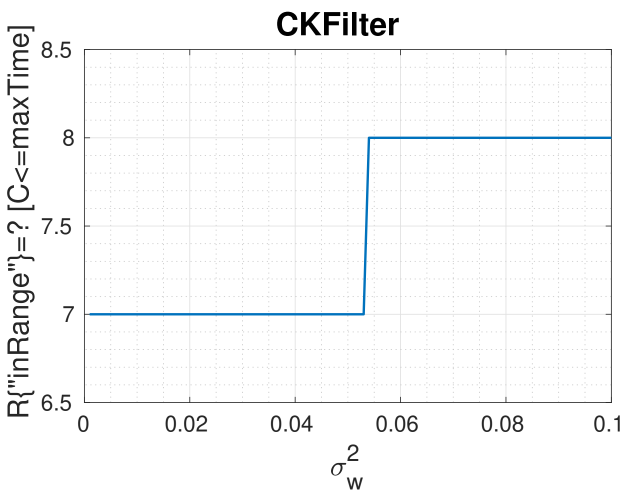


Figure 7.7: Resulted *inRange* for σ_w^2 values between 0.001 and 0.1.

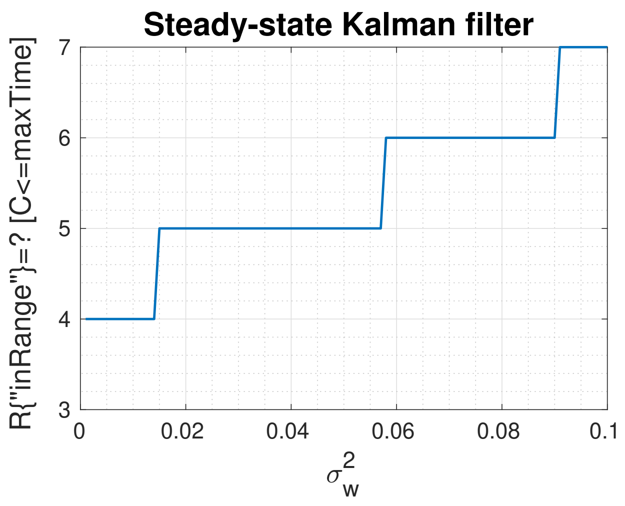


Figure 7.8: Resulted *inRange* for σ_w^2 values between 0.001 and 0.1.

the “quality” of the filters considered. Moreover, the `decPlaces` and `maxTime` values are set to 3 and 20, respectively. As explained in Section 5.3, for `maxTime`= 20, the value of the 95% lower bound we are seeking is 17.

In Figures 7.7 and 7.8, we note that for the minimum noise variance value considered, $\sigma_w^2 = 0.001$, the expected value of the *inRange* reward is 7 (i.e. $\approx$ 39%), and 4 (i.e. $\approx$ 22%), for the conventional and the steady-state Kalman filters, respectively.

However, these values fall far from the 95% bound we are seeking to satisfy. Moreover, as noise, in increments of 0.001, is injected in the two filters, the *inRange*

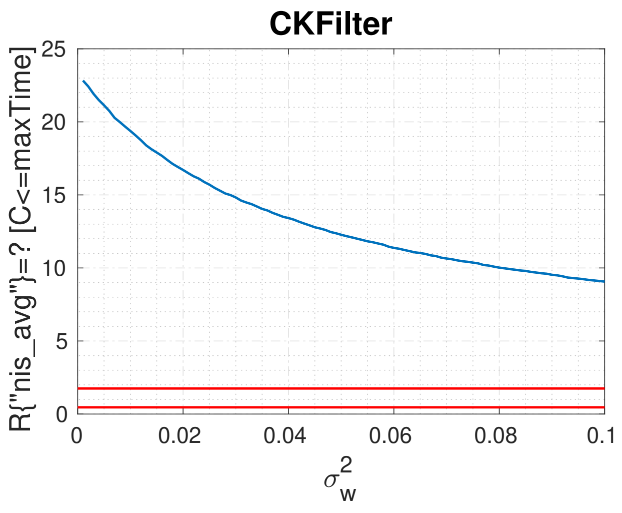


Figure 7.9: Resulted *nis_avg* for σ_w^2 values between 0.001 and 0.1.

reward values steadily increase, and reach a value of 8 and 7 for the conventional and steady-state Kalman filter, respectively. This happens when the process noise covariance matrices are constructed by setting a maximum value of σ_w^2, for this set of experiments, that is 0.1. The *inRange* reward values computed for the two filter variants, tell us that the number of times the value of the innovations falls within $\pm 2\sqrt{S_k}$, between a σ_w^2 0.001 and 0.1, has increased by 5.4% and by 17% for the two filters, approximately. The increase on the computed reward values of 17% for the steady-state Kalman filter, indicates that the effect of artificial noise injection is more prominent, compared to the equivalent increase of 5.4% for the `CKFilter`. However, the $\approx$ 44.4% and the $\approx$ 39% values, for the `CKFilter` and the steady-state filter, when $\sigma_w^2 = 0.1$, are still significantly less than the 95% bound which is required, for the two types of filters to be considered consistent. As a result the process noise covariance matrices, and consequently the filters constructed in this range should be rejected.

The second property we verify, shown in Figures 7.9 and 7.10 is whether the magnitude of the innovations is proportionate to the computed covariances of the conventional and of the steady-state Kalman filters. The y-axis of Figures 7.9 and 7.10 denotes the different values of the time-average normalised innovation squared

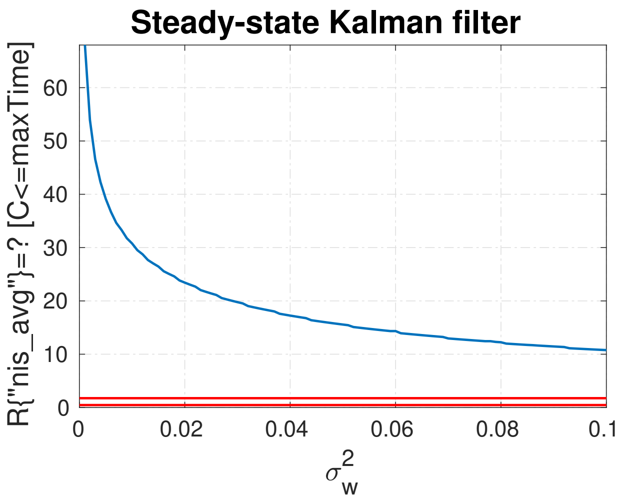


Figure 7.10: Resulted *nis_avg* for σ_w^2 values between 0.001 and 0.1.

statistic, which has been computed for different σ_w^2 values in the range of 0.001 to 0.1 inclusive. The goal of this experiment is to verify the consistency of the conventional and steady-state Kalman filters under different process noise covariance matrices, and choose a process noise covariance matrix which drives the aforementioned statistic to fall between the appropriate acceptance regions. Those regions are visualised as the lower and upper red lines on the plots, and essentially define the lower (0.46) and upper (1.75) bounds of the χ^2 distribution, respectively.

For the `CKFilter`, Figure 7.9, we note that as the value of the noise variance σ_w^2 increases, the *nis_avg* values decrease, implying that the consistency of the filter is increasing as more noise is being injected to the filter. This has been also observed on the first set of experiments. For example, for a $\sigma_w^2 = 0.001$ the respective time-average normalised innovation squared (*nis_avg*) reward value is 23, approximately. On the other extreme of the plot, when $\sigma_w^2 = 0.1$ the *nis_avg* reward value becomes $\approx$ 9, denoting an overall decrease, in the range of the σ_w^2 values considered, by a factor of 2.5.

A similar increasing trend, in terms of the improved consistency, is also observed for the steady-state Kalman filter, as shown in Figure 7.10. the *nis_avg* reward value is $\approx$ 69 for the minimum value of noise variance considered (e.g. $\sigma_w^2 = 0.001$).

The reward value of 69 is considerably larger compared to the reward value of 23, computed for the `CKFilter` for the same input. The larger reward of the steady-state Kalman filter can be explained by the fact that suboptimal Kalman gain is being used in the computations at every time step. This is in contrast to the `CKFilter` in which the optimal Kalman gain is being used. For example, for this particular case, the steady-state Kalman gain propagated through the Markov chain states is $[0.141, 0.009]^T$. On the other hand, the Kalman gain of the `CKFilter`, computed from the state ($s = 2$, $t = 2$), is $[0.666, 0.333]^T$ and converges to the values of $[0.183, 0.014]^T$ at state ($s = 20$, $t = 20$). In fact, the initial difference, when $\sigma_w^2 = 0.001$, between the *nis_avg* rewards of the two filters, is 46, falls to a value of less than 3 for $0.44 \leq \sigma_w^2 \leq 0.1$. The gradual decrease in the difference between the filters' rewards indicates that as more noise is being injected, their performance is becoming similar.

However, despite this decrease, all of the 200 different conventional and steady-state Kalman filters cannot be considered as consistent. This is because all of the noise covariance matrices, 100 for each filter type in this case, result in *nis_avg* values to fall considerably far from the 1.75 critical value. As a result, the null hypothesis, which states that the filter is consistent, should be rejected for both the conventional and the steady-state Kalman filters. In Figures 7.11 and 7.12, we show the combined results of the previous experiments, for a numerical precision of four, five, and six decimal places, respectively. In the first row of the respective figures, the *inRange* reward values are shown, while the second row depicts the results of the *nis_avg* reward structures, for the two filter variants. For the `CKFilter`, Figure 7.11, and for the steady-state filter, Figure 7.12, we note that an increase in the numerical precision does not have any significant effect on the accuracy of the verification result, for both of the reward structures. This can be observed by the tendency by which the *inRange* and the *nis_avg* values evolve over different σ_w^2 values, which is similar irrespective of the degrees of numerical precision used. In fact, for both

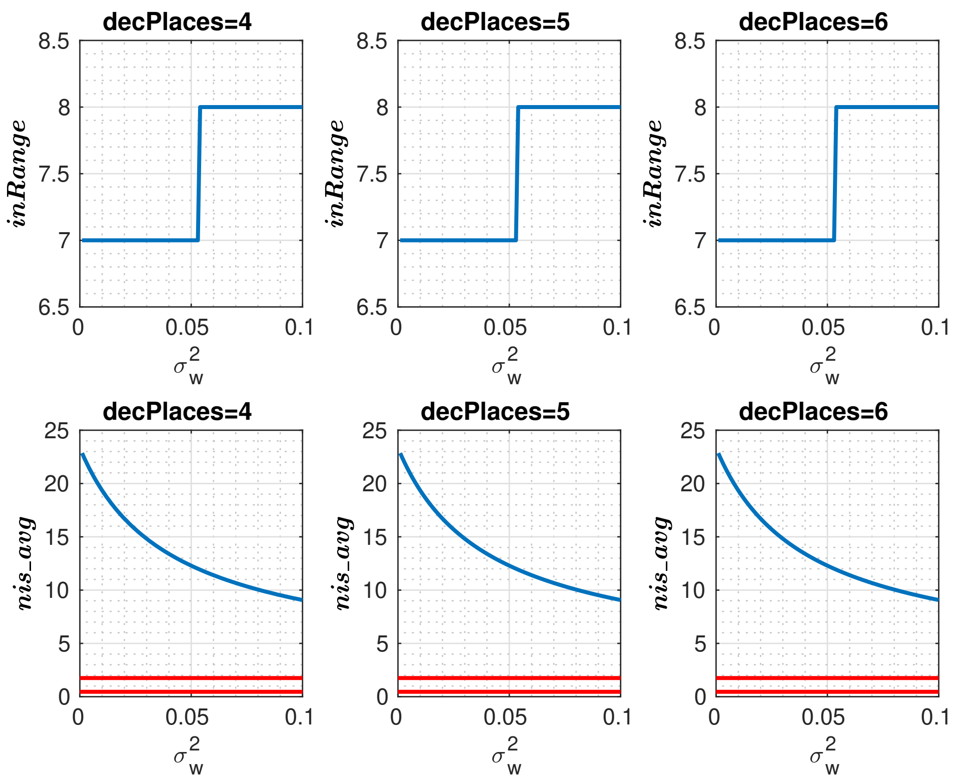


Figure 7.11: Resulted reward values for various degrees of numerical precision for the `CKFilter`.

of the filters, the average difference between the computed rewards from models constructed with `decPlaces`$=$ 4 and `decPlaces`$=$ 6, is zero. This means that, similar to the case where the `decPlaces`$=$ 3, the process noise covariance matrices, and consequently the filters, should be rejected.

In the second set of experiments, shown in Figures 7.13 and 7.14, the setup is similar to the first set of experiments, except that the σ_w^2 values, on the interval $[0.1, 5.5]$, are injected in larger increments of 0.1, instead of 0.001. In Figure 7.13, we show the results for the *inRange* reward values, which were computed for the specified interval, for the two filters. For the `CKFilter` in Figure 7.13, the *inRange* reward values reach a plateau at 16, for a value of σ_w^2 between 3.2 and 5.5, inclusively. Moreover, the *nis_avg* reward values, start to reside inside the confidence region for σ_w^2 values ≥ 3.5, for the specified interval. This means that on average, in 16 out of the 18 states of a particular path, the value of the innovation is expected

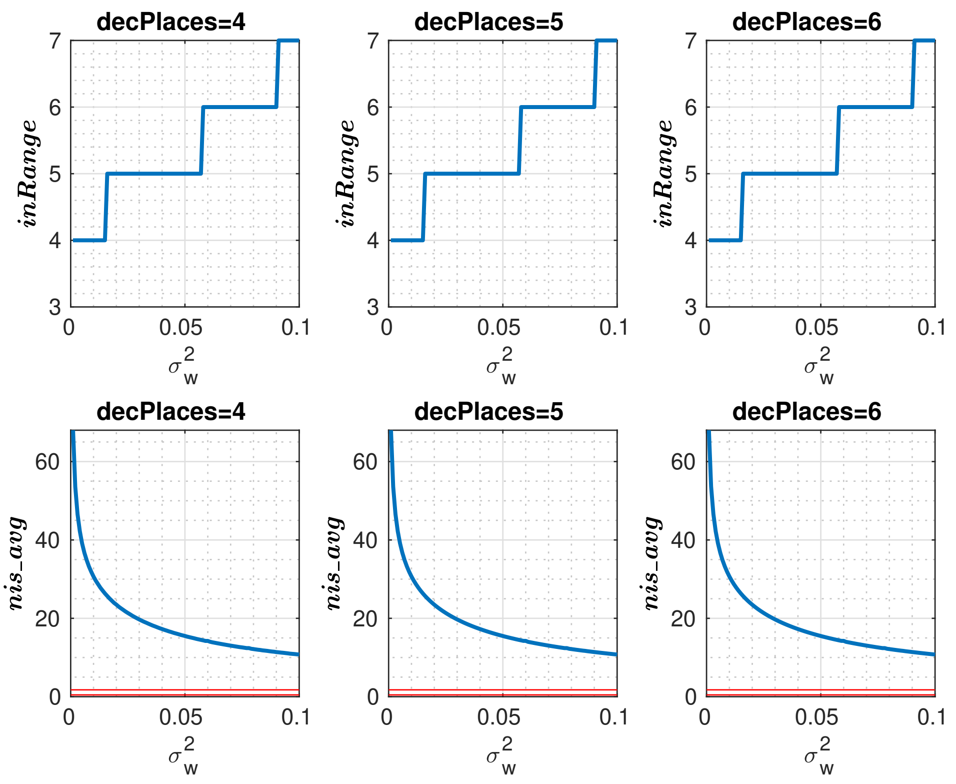


Figure 7.12: Resulted reward values for various degrees of numerical precision for the square-root Kalman filter.

to fall between two standard deviations of the mean. This gives us a value of $\approx$ 89%, which is less than the 95% bound we seek to satisfy. In consequence, the verification procedure of this particular consistency criterion was not successful, for the 55 different conventional Kalman filters which were verified for this particular range. On the other hand, 13 out of the 55 different steady-state Kalman filters verified, satisfied this particular consistency property, when their process noise covariance matrices Q were constructed with a σ_w^2 value between 4.3 and 5.5 inclusive.

In Figure 7.14, we show the results for the *inRange* reward values, which were computed for the specified interval, for the two filters. For example, the *nis_avg* value is $\approx$ 1.75 when verifying the `CKFilter` whose process noise covariance matrix, Q, has been constructed using a σ_w^2 value of 3.5. For the subsequent σ_w^2 values (e.g. $3.6, 3.7, ...$), the *nis_avg* values are constantly decreasing. For instance, for the maximum σ_w^2 value we consider (e.g. 5.5) , the computed *nis_avg* reward is 1.46.

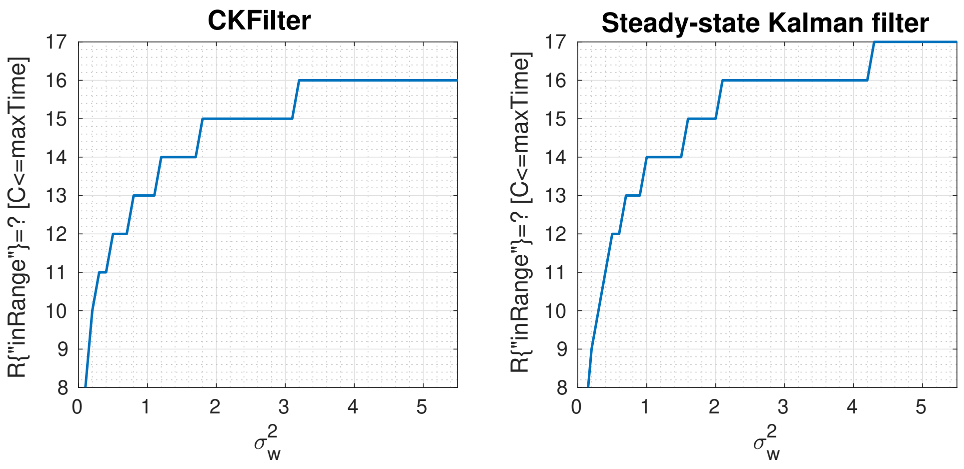


Figure 7.13: Resulted *inRange* for σ_w^2 values between 0.1 and 5.5 inclusive.

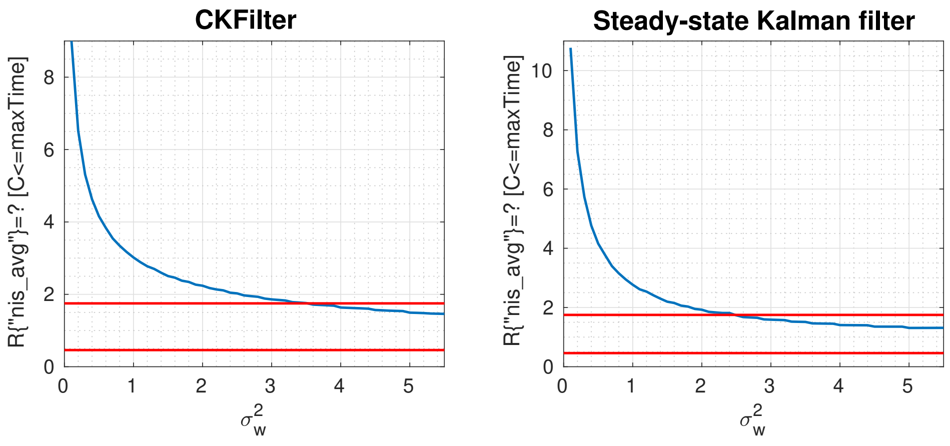


Figure 7.14: Resulted *nis_avg* for σ_w^2 values between 0.1 and 5.5 inclusive.

This means that the time-average normalised innovation squared quantity starts to stabilise itself when it passes the upper bound of the confidence region (e.g. $\sigma_w^2 \geq 3.6$). This results in the non-rejection of the null hypothesis, which states that the filter is consistent. In fact, all of the 21 different filters constructed with σ_w^2 values from 3.5 to 5.5 inclusive, should not be declared as inconsistent. On the other hand, while the *nis_avg* reward computed for the steady-state Kalman filter is initially much larger compared to the conventional one, it reaches the acceptance region faster. In fact, its *nis_avg* reward is 1.74 when it is constructed with a $\sigma_w^2 = 2.5$, which is ten times less than its conventional equivalent. It also indicates, that for this particular case, the performance of the steady-state Kalman filter surpasses the conventional filter's one, since less noise has to be injected to satisfy the consistency criterion.

7.3.3 Threats to Internal and External Validity

In terms of internal validity threats, first we consider the granularity level of the noise (`gLevel`) and its impact both on scalability and on the verification results. Note that we explicitly refer to verification results for properties which relate to modelling errors. This is because, as we have described in Section 5.1.2, numerical errors are not affected by the change in `gLevel`.

In the results presented, we assumed verifying Kalman filters mostly operating on real-time systems and processing real-time data. Since in these types of systems obtaining the verification result in a short amount of time is crucial, we focused more on scalability by setting the value of `gLevel` to 2. Closely related to scalability is the issue of accuracy and, as it has been noted in Section 5.1.2, the value of `gLevel` may affect the accuracy of the result. However, while it would have been easier, and potentially less accurate, not to include the stochastic system model in the DTMC, thus assuming measurements emitted from an unknown source, we chose to build a more expressive model which can be used for verification purposes; that is, a Markov chain whose trajectories capture the execution of a Kalman filter estimating the state of a stochastic dynamical system under a noisy measurement model.

With respect to external validity threats, the kinematic state model was proposed as the performance model for CPU utilisation, having taken into account other research works on the topic, as we presented in Section 7.2. However, the main focus of our work is on verification and, while it might seem as a reasonable model to track CPU utilisation data, more experimentation could potentially be needed to assess it in real situations. Furthermore, we did not consider Kalman filters specified in higher dimensions than two and we cannot claim that our work generalises in higher dimensions, for example, in cases where one would prefer to model each VM as a separate entity with a different performance model. However, as explained in Section 7.2, this was a design decision we consciously made for two reasons: i) for modelling a scenario which occurs in a real cloud, where the average CPU utilisation across the VMs in an auto-scaling group is emitted as a single measurement; ii) for efficiency, since by treating the auto-scaling group as a single entity the dimensionality of matrices and vectors remains constant, irrespective of whether VMs are added or removed.

Finally, in terms of the results presented in this chapter collectively, we generalised our approach, where possible, by considering four different filter variants and two kinematic state models. Also, we verified properties of 1852 different filter implementations, by taking into account the two classes of errors which affect Kalman filter implementations. Overall, for the two classes of errors (numerical and modelling) we verified, the tool and supporting files are available at [25] to enable reproducibility of the results.

7.4 Summary of Contributions

The contributions of this chapter can be summarised as follows:

- The successful demonstration of our tool VerFilter as reliable software, to be used for the verification of various types of Kalman filters. This involved the successful implementation of the novel verification techniques, which were presented and implemented in Chapters 5 and 6, respectively.

- A detailed evaluation which involved a rigorous quantitative analysis on the effectiveness of various Kalman filter implementations, using VerFilter. Overall, several properties of the conventional Kalman filter, the steady-state filter, Carlson-Schmidt square-root filter and the Bierman-Thornton U-D filter were verified. Overall, the numerical stability and consistency properties of 1852 different filter implementations were verified. The verification of the latter properties in particular were focused on cloud computing by modelling the CPU utilisation as a kinematic state model.

CHAPTER 8

Conclusions

8.1 Summary and Evaluation

The main aim of this work was to provide formal performance guarantees for key resource provisioning methods in cloud computing. In particular, we developed a novel framework based on quantitative verification methods which can be used for the automatic verification of automated resource provisioning mechanisms in the cloud.

We set out to investigate whether the gap that exists in the literature in terms of the verification of reactive resource provisioning approaches, could be bridged using probabilistic model checking. In Chapter 4 we presented a novel probabilistic verification scheme, followed by a validation on VMs rented from two major public cloud providers using two different cloud computing models, IaaS and PaaS, for Amazon EC2 and Microsoft Azure, respectively.

In particular, in Chapter 4 we have developed a Markov model using the PRISM model checker, in order to capture the dynamics of the auto-scaling process. This allowed us, for the Amazon EC2 case, to compute probabilities of CPU utilisation and response time violation for each auto-scaling policy given as an input to our model. Then, by using ROC analysis we were able to refine our original estimates, and find a global estimate which best represented a threshold for differentiating between auto-scaling policies which could be flagged as QoS violators or non-violators. Our experiments showed that our verification scheme can be of valuable assistance to cloud application owners and system administrators in formally configuring and verifying the auto-scaling policies of their applications/systems in the cloud.

For the Microsoft Azure case we focused on the temporal parameters of an auto-scaling policy, by considering cool-down periods of one and five minutes. Then, our goal was to show the usefulness of the PRISM model checker to compute the expected number of bad auto-scale decisions, and we presented our results having taken into account CPU utilisation as a performance metric. Moreover, we showed the difficulties that manifest in the modelling process as the cool-down period becomes shorter, and highlighted the difficult problem of attributing a (meaningful) duration to the states in the model in order to match to an extent the real situation.

We then proceeded to exploring the role quantitative verification, and PRISM especially, could play in modelling and verifying a certain class of resource provisioning approaches, which are based on state estimation, and Kalman filters in particular. This consisted of an in-depth investigation of research works in other fields, such as automotive and aerospace engineering, where Kalman filters have been extensively applied. This allowed us to lay the foundations to develop a general framework, as presented in Chapter 5, for modelling Kalman filter implementations and techniques to systematically construct a Markov model of the filter's operation using truncation and discretisation of the stochastic noise model. Then, we established consistency properties of the Kalman filter by translating them to temporal logic. Moreover, we considered two sources of errors which could render the Kalman filter inconsistent: i) numerical and ii) modelling errors. Specifically, for the numerical errors, we were concerned with the propagation of the estimation-error covariance matrix P in the time and measurement updates, since it is this the part of the filter that could hinder its numerical stability and so cause it to produce erroneous results. Furthermore, for the modelling errors, as a source of inconsistency, we studied how modelling error compensation techniques can be verified using probabilistic model checking.

Then, in Chapter 6 we presented our software implementation, VerFilter, which is built on top of PRISM and incorporates all of the techniques considered. VerFilter can currently handle the creation and verification of four different Kalman filter variants, based on inputs given by the user. This is, to the best of our knowledge, the first work where these types of problems are analysed in a probabilistic verification setting.

Finally, in Chapter 7 we put our software, VerFilter, in action through an extensive validation and evaluation process. This chapter also addressed the secondary aim of our work, which was to investigate the generalisability of our approach to other types of problems. We successfully demonstrated the reliability of our tool VerFilter, in terms of its verification ability on various types of Kalman filters. This was achieved by successfully implementing the novel verification techniques, which were presented and implemented in Chapters 5 and 6, respectively. Overall, several properties of the conventional Kalman filter, the steady-state filter, Carlson-Schmidt square root filter and the Bierman-Thornton U-D filter were verified.

8.2 Future Work

There are many possible areas in which the work in this thesis could be developed further. For example, for the quantitative verification of reactive resource provisioning mechanisms (e.g. auto-scaling policies), on Amazon EC2 we dealt mainly with the dynamics of an auto-scaling policy by varying the increments and the initial VMs in operation, as part of the controllable parameters. A potential direction for future work would be to analyse the effects of varying the other controllable parameters of an auto-scaling policy, such as the percentages of the scale-out and scale-in adjustments.

Moreover, for the Microsoft Azure case, the controllable parameters which were varied were the increments, the initial VMs in operation and the respective cool-down periods. A potential extension, which could be implemented in the future,is to experiment with cool-down periods of various durations, and conduct further research in the context of auto-scaling policies which dynamically change.

As far as the quantitative verification of Kalman filters is concerned, an improvement would be to consider models which are expressed in higher dimensional matrices. With respect to that, we would like to investigate how our work generalises not only in terms of various types of filters, but also in terms of matrices of higher dimensions. It would also be beneficial to perform a thorough performance analysis for these types of models to investigate the scalability of our techniques to see how it compares to the current scalability results.

Furthermore, we would like to investigate how our work can be extended to nonlinear models. For example, in our work the CPU utilisation model assumes that the incoming load is linear. It would be interesting to see how we can model CPU utilisation which corresponds to nonlinear load. This would potentially mean that we would need to incorporate in our framework a nonlinear version of the Kalman filter such as the Extended Kalman Filter. Lastly, we would like to research further how this work can be expanded to other linear algebra applications.

8.3 Conclusion

In conclusion, we have successfully answered our main research question which was whether we can use formal verification to produce formal performance guarantees for both control and estimation mechanisms in cloud computing. In particular, we developed novel approaches based on formal verification to produce performance guarantees on particular rule-based auto-scaling policies. Our experimental results show that the modelling process along with the model itself can be very effective in providing the necessary formal reasoning to cloud application owners with respect to the configuration of their auto-scaling policies, and consequently helping them to specify an auto-scaling policy which could minimise QoS violations.

Furthermore, we have successfully shown how quantitative verification can be used in resource provisioning contexts where Kalman filters are used to track several performance parameters. We have presented a framework for the modelling and verification of Kalman filter implementations. This framework is general enough to analyse a variety of different implementations and various system models, and to study a range of numerical and modelling error issues which may hinder the effective deployment of the filters in practice. We have created a software tool, VerFilter, implemented the techniques in it, and illustrated its applicability and scalability with a range of experiments.

APPENDIX A

Interfaces

This appendix shows the Java code for the interfaces which were presented in Sections 6.2.2 and 6.2.3, respectively. In particular, Section A.1 presents the methods available in PRISM's `ModelGenerator` interface. Section A.2 shows the methods available in VerFilter's `KalmanFilter` interface.

A.1 The ModelGenerator interface

Table A.1: The ModelGenerator interface.

Method	**Description**
`List<String> getVarNames()`	Returns a list with the names of the variables stored in the state.
`List<Type> getVarTypes()`	Returns a list with the types of the variables stored in the state.
`State getInitialState()`	Returns the initial state.
`int getNumChoices()`	Returns the number of nondeterministic choices.
`double getNumTransitions(int i)`	Returns the total number of transitions.
`double getTransProbability(int i,int offset)`	Returns the transition probability based on a given choice.
`void exploreState(State exploreState)`	Explores the outgoing transitions of the current state.
`State getExploreState()`	Returns the state that is being explored.
`State computeTransTarget(int i,int offset)`	Returns the state of a transition.
`int getNumLabels()`	Returns the number of different labels.
`List<String> getLabelNames()`	Returns a list with the names of the labels.
`boolean isLabelTrue(int i)`	Returns true or false depending on the label's ith value.
`double getStateReward(int r, State state)`	Returns the rth reward of the state.

A.2 The `KalmanFilter` interface

Listing A.1: Some of the methods in the `KalmanFilter` interface

```
public interface KalmanFilter {
  public void predict();
  public void correct(RealVector z);
  public int getStateDimension();
  public int getMeasurementDimension();
  public RealVector getStateEstimationVector();
  public RealMatrix getErrorCovarianceMatrix();
  public RealVector getInnovation();
  public RealMatrix getInnovationCovariance();
}
```

List of References

[1] "Dynamic Scaling - Auto Scaling," http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scale-based-on-demand.html#as-scaling-adjustment, 2016.

[2] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut, "Using machine learning for black-box autoscaling," in *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, Nov. 2016, pp. 1–8.

[3] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From Data Center Resource Allocation to Control Theory and Back," in *2010 IEEE 3rd International Conference on Cloud Computing*, Jul. 2010, pp. 410–417.

[4] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight Resource Scaling for Cloud Applications," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012, pp. 644–651.

[5] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal Autoscaling in a IaaS Cloud," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 173–178.

[6] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically Scaling Applications in the Cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, Jan. 2011.

[7] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *2011 IEEE International Conference on Cloud Computing (CLOUD)*, Jul. 2011, pp. 716–723.

[8] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, "Dependable Horizontal Scaling Based on Probabilistic Model Checking," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2015, pp. 31–40.

[9] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking Time-varying Parameters in Software Systems with Extended Kalman Filters," in *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '05. Toranto, Ontario, Canada: IBM Press, 2005, pp. 334–345.

[10] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters," in *Proceedings of the 6th International Conference on Autonomic Computing*, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 117–126. [Online]. Available: http://doi.acm.org/10.1145/1555228.1555261

[11] T. Charalambous and E. Kalyvianaki, "A min-max framework for CPU resource provisioning in virtualized servers using H-infinity filters," in *49th IEEE Conference on Decision and Control (CDC)*, Dec 2010, pp. 3778–3783.

[12] R. E. Kalman, "A new approach to linear filtering and prediction problems," *ASME Journal of Basic Engineering*, 1960.

[13] P. S. Maybeck, *Stochastic models, estimation, and control: Volume 1*, ser. Mathematics in science and engineering. Burlington, MA: Elsevier Science, 1982.

[14] M. S. Grewal and A. P. Andrews, *Kalman Filtering: Theory and Practice with MATLAB*, 4th ed. Wiley-IEEE Press, 2014.

[15] D. Simon, *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley-Interscience, 2006.

[16] B. P. Gibbs, *Advanced Kalman Filtering, Least-Squares and Modeling: A Practical Handbook*. John Wiley & Sons, Inc., 2011.

[17] P. Zarchan and H. Musoff, *Fundamentals of Kalman filtering : a practical approach*, 4th ed. Reston, VA: American Institute of Aeronautics and Astronautics, 2015.

[18] M. Kwiatkowska, G. Norman, and D. Parker, “Stochastic Model Checking,” in *Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation*, ser. SFM’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 220–270.

[19] ——, “PRISM 4.0: Verification of probabilistic real-time systems,” in *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[20] “Supporting material,” www.prismmodelchecker.org/files/fgcs-autoscaling/.

[21] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, “Automated verification techniques for probabilistic systems,” in *Formal Methods for Eternal Networked Software Systems (SFM’11)*, ser. LNCS, M. Bernardo and V. Issarny, Eds., vol. 6659. Springer, 2011, pp. 53–113.

[22] H. Hansson and B. Jonsson, “A logic for reasoning about time and reliability,” *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[23] Y. Bar-Shalom and X.-R. Li, *Estimation with Applications to Tracking and Navigation.* New York, NY, USA: John Wiley & Sons, Inc., 2001.

[24] X. R. Li and V. P. Jilkov, “Survey of maneuvering target tracking. part i. dynamic models,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39, no. 4, pp. 1333–1364, Oct 2003.

[25] “Supporting material,” www.prismmodelchecker.org/files/fm19kf/.

[26] A. Evangelidis, D. Parker, and R. Bahsoon, “Performance modelling and verification of cloud-based auto-scaling policies,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 355–364.

[27] ——, “Performance modelling and verification of cloud-based auto-scaling policies,” *Future Generation Computer Systems*, vol. 87, pp. 629 – 638, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17312475

[28] A. Evangelidis and D. Parker, "Quantitative verification of numerical stability for Kalman filters," in *Proc. 23rd International Symposium on Formal Methods (FM'19)*, ser. LNCS, vol. 11800. Springer, 2019, pp. 425–441.

[29] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer, "Verifying Cloud Services: Present and Future," *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 2, pp. 6–19, Jul. 2013.

[30] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *J. Netw. Syst. Manage.*, vol. 23, no. 3, p. 567–619, Jul. 2015. [Online]. Available: https://doi.org/10.1007/s10922-014-9307-7

[31] G. Galante and L. de Bona, "A Survey on Cloud Computing Elasticity," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing (UCC)*, Nov. 2012, pp. 263–270.

[32] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," 2013, pp. 23–27.

[33] W. Sellami, H. Kacem, and A. Kacem, "Elastic Multi-tenant Business Process Based Service Pattern in Cloud Computing," in *2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2014, pp. 154–161.

[34] S. Boyd and L. Vandenberghe, *Introduction to Applied Linear Algebra*, ser. Technology and Engineering. Cambridge University Press, 2018.

[35] B. Anderson and J. Moore, *Optimal Filtering*, ser. Dover Books on Electrical Engineering. Dover Publications, 2012. [Online]. Available: https://books.google.co.uk/books?id=iYMqLQp49UMC

[36] S. Boyd, "Lecture notes for ee263, introduction to linear dynamical systems," September 2008.

[37] T. Duriez, S. L. Brunton, and B. R. Noack, *Machine Learning Control - Taming Nonlinear Dynamics and Turbulence*, 1st ed. Springer Publishing Company, Incorporated, 2016.

[38] C. Moler and C. V. Loan, "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later," *SIAM REVIEW*, vol. 45, no. 1, pp. 801–836, 2003.

[39] V. P. J. X. Rong Li, "Survey of maneuvering target tracking: dynamic models," pp. 4048 – 4048 – 24, 2000. [Online]. Available: https://doi.org/10.1117/12.391979

[40] S. S. Blackman and a. Popoli, Robert, *Design and analysis of modern tracking systems.* Boston : Artech House, 1999, includes bibliographical references and index.

[41] Y. Bar-Shalom, *Tracking and Data Association.* San Diego, CA, USA: Academic Press Professional, Inc., 1987.

[42] C.-B. Chang and K.-P. Dunn, *Applied State Estimation and Association.* Cambridge, Massachusetts: The MIT Press, 2016.

[43] R. G. Brown and P. Y. C. Hwang, *Introduction to Random Signals and Applied Kalman Filtering with MATLAB Exercises, 4th Edition.* Wiley Global Education, Jan. 2012.

[44] M. Vetterli, J. Kovačević, and V. K. Goyal, *Foundations of Signal Processing*, 3rd ed. Cambridge University Press, 2014.

[45] A. Jazwinski, *Stochastic Processes and Filtering Theory*, ser. Mathematics in Science and Engineering. Elsevier Science, 1970. [Online]. Available: https://books.google.co.uk/books?id=nGlSNvKyY2MC

[46] R. Kalman, "On the general theory of control systems," *IRE Transactions on Automatic Control*, vol. 4, no. 3, pp. 110–110, December 1959.

[47] B. Southall, B. F. Buxton, and J. A. Marchant, "Controllability and observability: Tools for Kalman filter design," in *Proceedings of the British Machine Vision Conference 1998, BMVC 1998, Southampton, UK, 1998*, 1998, pp. 1–10.

[48] D. J. Clements and B. D. O. Anderson, *Singular optimal control : the linear-quadratic problem / David J. Clements, Brian D. O. Anderson.* Springer-Verlag Berlin ; New York, 1978.

[49] J. Diard, P. Bessière, and E. Mazer, "A survey of probabilistic models, using the bayesian programming methodology as a unifying framework," in *The*

Second International Conference on Computational Intelligence, Robotics and Autonomous Systems CIRAS, 2003.

[50] J. L. Speyer, *Stochastic Processes, Estimation, and Control.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008.

[51] R. Labbe, "Kalman and bayesian filters in python," https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python, 2014.

[52] D. Bertsekas and J. Tsitsiklis, *Introduction to Probability*, ser. Athena Scientific optimization and computation series. Athena Scientific, 2008. [Online]. Available: https://books.google.co.uk/books?id=yAy-PQAACAAJ

[53] G. Strang and K. Borre, *Linear Algebra, Geodesy, and GPS.* Wellesley, MA, USA: Wellesley-Cambridge Press, 1997.

[54] P. Kaminski, A. Bryson, and S. Schmidt, "Discrete square root filtering: A survey of current techniques," *IEEE Transactions on Automatic Control*, vol. 16, no. 6, pp. 727–736, December 1971.

[55] R. H. Battin, *Astronautical guidance*, ser. Electronic sciences. McGraw-Hill, 1964.

[56] N. A. Carlson, "Fast triangular formulation of the square root filter," *AIAA Journal*, vol. 11, no. 9, pp. 1259–1265, 1973.

[57] C. L. Thornton, "Triangular covariance factorizations for Kalman filtering," Ph.D. dissertation, University of California at Los Angeles, 1976.

[58] C. D'Souza and R. Zanetti, "Information formulation of the UDU Kalman filter," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 1–1, 2018.

[59] G. J. Bierman, "Measurement updating using the u-d factorization," in *1975 IEEE Conference on Decision and Control including the 14th Symposium on Adaptive Processes*, Dec 1975, pp. 337–346.

[60] G. J. Bierman, *Factorization Methods for Discrete Sequential Estimation.* Elsevier Science, 1977.

[61] G. Frison and J. B. Jørgensen, "Efficient implementation of the Riccati recursion for solving linear-quadratic control problems," in *2013 IEEE International Conference on Control Applications (CCA)*, Aug 2013, pp. 1117–1122.

[62] F. L. Lewis, L. Xie, and D. Popa, *Optimal and Robust Estimation: With an Introduction to Stochastic Control Theory.* CRC Press : Boca Raton, 2007.

[63] D. Bini, B. Iannazzo, and B. Meini, *Numerical Solution of Algebraic Riccati Equations.* Society for Industrial and Applied Mathematics, 2011. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611972092

[64] E. D. Denman and A. N. Beavers, "The matrix sign function and computations in systems," *Applied Mathematics and Computation*, vol. 2, no. 1, pp. 63 – 94, 1976. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0096300376900205

[65] D. Kleinman, "On an iterative technique for Riccati equation computations," *IEEE Transactions on Automatic Control*, vol. 13, no. 1, pp. 114–115, February 1968.

[66] G. Hewer, "An iterative technique for the computation of the steady state gains for the discrete optimal regulator," *IEEE Transactions on Automatic Control*, vol. 16, no. 4, pp. 382–384, August 1971.

[67] A. J. Laub, "A schur method for solving algebraic Riccati equations," in *1978 IEEE Conference on Decision and Control including the 17th Symposium on Adaptive Processes*, Jan 1978, pp. 60–65.

[68] W. F. Arnold and A. J. Laub, "Generalized eigenproblem algorithms and software for algebraic Riccati equations," *Proceedings of the IEEE*, vol. 72, no. 12, pp. 1746–1754, Dec 1984.

[69] M. Kwiatkowska, "Quantitative verification: Models, techniques and tools," in *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE).* ACM Press, September 2007, pp. 449–458.

[70] M. Ujma, "On verification and controller synthesis for probabilistic systems at runtime," PhD thesis, University of Oxford, 2015.

[71] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic Model Checking," in *Formal Methods for Performance Evaluation*, ser. Lecture Notes in Computer Science, M. Bernardo and J. Hillston, Eds. Springer Berlin Heidelberg, May 2007, no. 4486, pp. 220–270.

[72] G. Norman and D. Parker, "Quantitative verification: Formal guarantees for timeliness, reliability and performance," The London Mathematical Society and the Smith Institute, Tech. Rep., 2014.

[73] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu, "Resource provisioning for cloud computing," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '09. Riverton, NJ, USA: IBM Corp., 2009, pp. 101–111. [Online]. Available: https://doi.org/10.1145/1723028.1723041

[74] M. Macías, G. Smith, O. Rana, J. Guitart, and J. Torres, "Enforcing Service Level Agreements Using an Economically Enhanced Resource Manager," in *Economic Models and Algorithms for Distributed Systems*, ser. Autonomic Systems, D. Neumann, M. Baker, J. Altmann, and O. Rana, Eds. Birkhäuser Basel, 2009, pp. 109–127.

[75] J. Guitart, M. Macías, O. Rana, P. Wieder, R. Yahyapour, and W. Ziegler, "SLA-Based Resource Management and Allocation," in *Market-Oriented Grid and Utility Computing*, R. Buyya and K. Bubendorfer, Eds. John Wiley & Sons, Inc., 2009, pp. 261–284.

[76] N. J. Dingle, W. J. Knottenbelt, and L. Wang, "Service level agreement specification, compliance prediction and monitoring with performance trees," in *ESM 2008 - 2008 European Simulation and Modelling Conference: Modelling and Simulation 2008*, 2008.

[77] W. Dawoud, I. Takouna, and C. Meinel, "Elastic VM for Cloud Resources Provisioning Optimization," in *Advances in Computing and Communications*, ser. Communications in Computer and Information Science, A. Abraham, J. L. Mauri, J. F. Buford, J. Suzuki, and S. M. Thampi, Eds. Springer Berlin Heidelberg, Jul. 2011, no. 190, pp. 431–445.

[78] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya, "The Aneka Platform and QoS-driven Resource Provisioning for Elastic Applications on Hybrid Clouds," *Future Gener. Comput. Syst.*, vol. 28, no. 6, pp. 861–870, Jun. 2012.

[79] A. Freitas, N. Parlavantzas, and J.-L. Pazat, “Cost Reduction through SLA-driven Self-Management,” in *2011 Ninth IEEE European Conference on Web Services (ECOWS)*, Sep. 2011, pp. 117–124.

[80] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, “Efficient Resource Provisioning in Compute Clouds via VM Multiplexing,” in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC ’10. New York, NY, USA: ACM, 2010, pp. 11–20.

[81] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, “CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems,” in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC ’11. New York, NY, USA: ACM, 2011, pp. 5:1–5:14.

[82] M. Dhingra, J. Lakshmi, S. Nandy, C. Bhattacharyya, and K. Gopinath, “Elastic Resources Framework in IaaS, Preserving Performance SLAs,” in *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, Jun. 2013, pp. 430–437.

[83] L. Wu, S. Garg, S. Versteeg, and R. Buyya, “SLA-Based Resource Provisioning for Hosted Software-as-a-Service Applications in Cloud Computing Environments,” *IEEE Transactions on Services Computing*, vol. 7, no. 3, pp. 465–485, Jul. 2014.

[84] B. Nandi, A. Banerjee, S. Ghosh, and N. Banerjee, “Dynamic SLA based elastic cloud service management: A SaaS perspective,” in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, May 2013, pp. 60–67.

[85] L. Wu, S. Garg, and R. Buyya, “SLA-Based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments,” in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2011, pp. 195–204.

[86] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, “Cloud elasticity using probabilistic model checking,” *arXiv:1405.4699 [cs]*, May 2014.

[87] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, “AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers,” *ACM Trans. Comput. Syst.*, vol. 30, no. 4, pp. 14:1–14:26, Nov. 2012.

[88] S. A. Baset, “Cloud SLAs: Present and Future,” *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 2, pp. 57–66, Jul. 2012.

[89] A. Gandhi, P. Dube, A. Karve, A. P. Kochut, and L. Zhang, “Providing performance guarantees for cloud-deployed applications,” *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[90] J. C. Mogul, “Emergent (Mis)Behavior vs. Complex Software Systems,” in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys ’06. New York, NY, USA: ACM, 2006, pp. 293–304.

[91] S. Elbaum and D. S. Rosenblum, “Known Unknowns: Testing in the Presence of Uncertainty,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 833–836.

[92] G. Candea, S. Bucur, and C. Zamfir, “Automated Software Testing As a Service,” in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC ’10. New York, NY, USA: ACM, 2010, pp. 155–160.

[93] C. R. L. Neto and V. C. Garcia, “Cloud Testing Framework,” in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE ’13. New York, NY, USA: ACM, 2013, pp. 252–255.

[94] J. Zhou, S. Li, Z. Zhang, and Z. Ye, “Position Paper: Cloud-based Performance Testing: Issues and Challenges,” in *Proceedings of the 2013 International Workshop on Hot Topics in Cloud Services*, ser. HotTopiCS ’13. New York, NY, USA: ACM, 2013, pp. 55–62.

[95] L. Riungu, O. Taipale, and K. Smolander, “Research Issues for Software Testing in the Cloud,” in *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov. 2010, pp. 557–564.

[96] ——, “Software Testing as an Online Service: Observations from Practice,” in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, Apr. 2010, pp. 418–423.

[97] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, “Testing as a Service (TaaS) on Clouds,” in *2013 IEEE 7th International Symposium on Service Oriented System Engineering (SOSE)*, Mar. 2013, pp. 212–223.

[98] S. Huang, X. Xu, Y. Xiao, and W. Wang, "Cloud Based Test Coverage Service," in *2012 IEEE 19th International Conference on Web Services (ICWS)*, Jun. 2012, pp. 648–649.

[99] H. Srikanth and M. Cohen, "Regression testing in Software as a Service: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 372–381.

[100] B. Sengupta and A. Roychoudhury, "Engineering Multi-tenant Software-as-a-service Systems," in *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems*, ser. PESOS '11. New York, NY, USA: ACM, 2011, pp. 15–21.

[101] J. Kabbedijk and S. Jansen, "The Role of Variability Patterns in Multi-tenant Business Software," in *Proceedings of the WICSA/ECSA 2012 Companion Volume*, ser. WICSA/ECSA '12. New York, NY, USA: ACM, 2012, pp. 143–146.

[102] S. T. Ruehl and U. Andelfinger, "Applying Software Product Lines to Create Customizable Software-as-a-service Applications," in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, ser. SPLC '11. New York, NY, USA: ACM, 2011, pp. 16:1–16:4.

[103] Y.-H. Tung, S.-S. Tseng, and Y.-Y. Kuo, "A testing-based approach to SLA evaluation on cloud environment," in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, Aug. 2015, pp. 495–498.

[104] M. Palacios, J. García-Fanjul, J. Tuya, and C. De La Riva, "A Proactive Approach to Test Service Level Agreements," in *2010 Fifth International Conference on Software Engineering Advances (ICSEA)*, Aug. 2010, pp. 453–458.

[105] M. Palacios, J. Garcia-Fanjul, J. Tuya, and G. Spanoudakis, "Identifying Test Requirements by Analyzing SLA Guarantee Terms," in *2012 IEEE 19th International Conference on Web Services (ICWS)*, Jun. 2012, pp. 351–358.

[106] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, "Search-based Testing of Service Level Agreements," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1090–1097.

[107] M. A. S. Netto, C. Cardonha, R. L. F. Cunha, and M. D. Assuncao, "Evaluating Auto-scaling Strategies for Cloud Computing Environments," in *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, Sep. 2014, pp. 187–196.

[108] A. V. Papadopoulos, A. Ali-Eldin, K.-E. Årzén, J. Tordsson, and E. Elmroth, "Peas: A performance evaluation framework for auto-scaling strategies in cloud applications," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 4, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2930659

[109] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 75–86. [Online]. Available: https://doi.org/10.1145/3030207.3030214

[110] F. L. Ferraris, D. Franceschelli, M. P. Gioiosa, D. Lucia, D. Ardagna, E. Di Nitto, and T. Sharif, "Evaluating the auto scaling performance of Flexiscale and Amazon ec2 clouds," in *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012, pp. 423–429.

[111] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[112] M. Kwiatkowska, G. Norman, and D. Parker, "Advances and challenges of probabilistic model checking," in *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2010, pp. 1691–1698.

[113] S. Kikuchi and Y. Matsumoto, "Performance Modeling of Concurrent Live Migration Operations in Cloud Computing Systems Using PRISM Probabilistic Model Checker," in *2011 IEEE International Conference on Cloud Computing (CLOUD)*, Jul. 2011, pp. 49–56.

[114] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue, "Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples," in *Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems*, ser. QEST '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 299–308.

[115] V. Shmatikov, "Probabilistic Analysis of an Anonymity System," *J. Comput. Secur.*, vol. 12, no. 3,4, pp. 355–377, May 2004.

[116] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn, "Probabilistic Model Checking of Complex Biological Pathways," in *Computational Methods in Systems Biology*, ser. Lecture Notes in Computer Science, C. Priami, Ed. Springer Berlin Heidelberg, Oct. 2006, no. 4210, pp. 32–47.

[117] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive Software Needs Quantitative Verification at Runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, Sep. 2012.

[118] J. Cámara, W. Peng, D. Garlan, and B. Schmerl, "Reasoning about sensing uncertainty and its reduction in decision-making for self-adaptation," *Science of Computer Programming*, vol. 167, pp. 51 – 69, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642318302521

[119] J. Cámara, G. A. Moreno, and D. Garlan, "Stochastic Game Analysis and Latency Awareness for Proactive Self-adaptation," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS 2014. New York, NY, USA: ACM, 2014, pp. 155–164.

[120] ——, "Reasoning about human participation in self-adaptive systems," in *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 146–156. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821357.2821384

[121] R. Calinescu, K. Johnson, Y. Rafiq, S. Gerasimou, G. C. Silva, and S. N. Pehlivanov, "Continual verification of non-functional properties in cloud-based systems," in *NiM-ALP@MoDELS*, 2013.

[122] V. A. d. S. Júnior and S. Tahar, "Time Performance Formal Evaluation of Complex Systems," in *Formal Methods: Foundations and Applications*, ser. Lecture Notes in Computer Science, M. Cornélio and B. Roscoe, Eds. Springer International Publishing, Sep. 2015, pp. 162–177.

[123] E. Kalyvianaki, T. Charalambous, and S. Hand, "Adaptive Resource Provisioning for Virtualized Servers Using Kalman Filters," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 2, pp. 10:1–10:35, Jul. 2014.

[124] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *11th International*

Conference on Autonomic Computing (ICAC 14). Philadelphia, PA: USENIX Association, 2014, pp. 57–64. [Online]. Available: https://www.usenix.org/conference/icac14/technical-sessions/presentation/gandhi

[125] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Modeling the impact of workload on cloud resource scaling," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, Oct 2014, pp. 310–317.

[126] F. M. Boland and H. Nicholson, "Control of divergence in Kalman filters," *Electronics Letters*, vol. 12, no. 15, pp. 367–369, July 1976.

[127] A. Gelb and T. A. S. Corporation, *Applied Optimal Estimation*. The MIT Press, 1974.

[128] M. Verhaegen and P. V. Dooren, "Numerical aspects of different Kalman filter implementations," *IEEE Transactions on Automatic Control*, vol. 31, no. 10, pp. 907–917, October 1986.

[129] T. Kailath, *Linear Systems*. Englewood Cliffs, N.J: Prentice-Hall, 1980.

[130] R. S. Bucy and P. D. Joseph, *processes with applications to guidance*. Interscience Publishers New York, 1968.

[131] S. C. Chapra and R. Canale, *Numerical Methods for Engineers*, 5th ed. New York, NY, USA: McGraw-Hill, Inc., 2006.

[132] G. Abdelnour, S. Chand, S. Chiu, and T. Kido, "On-line detection and correction of Kalman filter divergence by fuzzy logic," in *1993 American Control Conference*, June 1993, pp. 1835–1839.

[133] Q. Ge, T. Shao, Z. Duan, and C. Wen, "Performance analysis of the Kalman filter with mismatched noise covariances," *IEEE Transactions on Automatic Control*, vol. 61, no. 12, pp. 4014–4019, Dec 2016.

[134] P. S. Maybeck, *Stochastic Models: Estimation and Control: Volume 2*, ser. Mathematics in Science and Engineering. Elsevier Science, 1982.

[135] D. Simon, "Kalman filtering with state constraints: A survey of linear and nonlinear algorithms," *IET Control Theory Applications*, vol. 4, no. 8, pp. 1303–1318, Aug. 2010.

[136] M. Moulin, L. Gluhovsky, and E. Bendersky, "Formal verification of maneuvering target tracking," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2003.

[137] G. Roşu, R. P. Venkatesan, J. Whittle, and L. Leuştean, *Certifying Optimality of State Estimation Programs.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 301–314. [Online]. Available: https://doi.org/10.1007/978-3-540-45069-6_30

[138] J. Whittle and J. Schumann, "Automating the implementation of Kalman filter algorithms," *ACM Trans. Math. Softw.*, vol. 30, no. 4, pp. 434–453, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1039813.1039816

[139] J. V. B. R. Gamboa, J. Cowles, "On the verification of synthesized Kalman filters," in *4th International Workshop on the ACL2 Theorem Prover and Its Applications.*, 2003.

[140] "Apache JMeter - Apache JMeter™," http://jmeter.apache.org/, 2016.

[141] "Best practices for autoscale - Microsoft Azure," https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/insights-autoscale-best-practices/, 2017.

[142] C. Romesburg, *Cluster Analysis for Researchers.* Lulu Press, 2004.

[143] H. Wang and M. Song, "Ckmeans.1d.dp: Optimal k-means Clustering in One Dimension by Dynamic Programming," *The R journal*, vol. 3, no. 2, pp. 29–33, Dec. 2011. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC5148156/

[144] D. Parker, "Implementation of symbolic model checking for probabilistic systems," Ph.D. dissertation, University of Birmingham, 2002.

[145] M. S. Pepe, "Receiver operating characteristic methodology," *Journal of the American Statistical Association*, vol. 95, no. 449, pp. 308–311, 2000. [Online]. Available: http://www.jstor.org/stable/2669554

[146] "Google cloud platform - Python docs samples," https://github.com/GoogleCloudPlatform/python-docs-samples/, 2016.

[147] "Amazon CloudWatch - Cloud & Network Monitoring Services," https://aws.amazon.com/cloudwatch/, 2016.

[148] B. Furht, "Cloud Computing Fundamentals," in *Handbook of Cloud Computing*, B. Furht and A. Escalante, Eds. Springer US, 2010, pp. 3–19.

[149] "JMeter-Plugin," https://jmeter-plugins.org/wiki/UltimateThreadGroup/, 2016.

[150] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, 1st ed. New York, NY, USA: Cambridge University Press, 2013.

[151] T. Fawcett, "An Introduction to ROC Analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[152] W. J. Krzanowski and D. J. Hand, *ROC Curves for Continuous Data*, 1st ed. Chapman & Hall/CRC, 2009.

[153] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica Et Biophysica Acta*, vol. 405, no. 2, pp. 442–451, Oct. 1975.

[154] "Microsoft azure app service - Bakery Web App Template," https://github.com/azure-appservice-samples/BakeryTemplate, 2017.

[155] N. Huber, S. Becker, C. Rathfelder, J. Schweflinghaus, and R. H. Reussner, "Performance modeling in industry: A case study on storage virtualization," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, May 2010, pp. 1–10.

[156] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer, 2014.

[157] N. L. Johnson, S. Kotz, and N. Balakrishnan, *Continuous univariate distributions.* New York: Wiley, 1994.

[158] J. Burkardt, “The truncated normal distribution,” 2014, Working paper.

[159] L. Wasserman, *All of Statistics: A Concise Course in Statistical Inference.* Springer Publishing Company, Incorporated, 2010.

[160] I. Reid, “Estimation ii,” 2001. [Online]. Available: http://www.robots.ox.ac.uk/~ian/Teaching/Estimation/LectureNotes2.pdf

[161] “Math - Commons-Math: The Apache Commons Mathematics Library.” [Online]. Available: http://commons.apache.org/math/

[162] G. W. Stewart, *Matrix Algorithms*, 1st ed. SIAM: Society for Industrial and Applied Mathematics, 1998.

[163] E. Makridis, K. M. Deliparaschos, E. Kalyvianaki, and T. Charalambous, “Dynamic cpu resource provisioning in virtualized servers using maximum correntropy criterion Kalman filters,” in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2017, pp. 1–8.

[164] F. Black and M. Scholes, “The pricing of options and corporate liabilities,” *Journal of Political Economy*, vol. 81, no. 3, pp. 637–54, 1973. [Online]. Available: https://EconPapers.repec.org/RePEc:ucp:jpolec:v:81:y:1973:i:3:p:637-54

[165] R. N. Mantegna and H. E. Stanley, “Stock market dynamics and turbulence: parallel analysis of fluctuation phenomena,” *Physica A: Statistical Mechanics and its Applications*, vol. 239, no. 1, pp. 255 – 266, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378437196004840

[166] ——, *Introduction to Econophysics: Correlations and Complexity in Finance.* Cambridge University Press, 1999.