FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Agile Software Performance Engineering

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Johannes Artner
Matrikelnummer 1127256

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Privatdoz. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Wien, 1. Dezember 2016

Johannes Artner

Manuel Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Agile Software Performance Engineering

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Johannes Artner
Registration Number 1127256

to the Faculty of Informatics
at the TU Wien
Advisor: Privatdoz. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Vienna, 1st December, 2016

Johannes Artner

Manuel Wimmer

Erklärung zur Verfassung der Arbeit

Johannes Artner
Wiedner Hauptstraße 121/10, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Dezember 2016

Johannes Artner

Kurzfassung

Performanz, unter dem Gesichtspunkt zeitlicher Anfrage-Antwortintervalle, ist eine charakteristische Eigenschaft von Softwaresystemen. Zur Evaluierung und Optimierung der Performanz gibt es in der wissenschaftlichen Literatur eine Vielzahl an Ansätzen, wobei aufgrund des zumeist sehr hohen Zusatzaufwandes und der zusätzlichen Komplexität die meisten dieser Ansätze wenig Verwendung in der Praxis finden. Das adäquate Management von Performanz anhand ingenieurmäßiger Prinzipien ist in der Praxis demfolgend sehr selten. Die Konzeption und Evaluation der Performanz von Softwaresystemen wird in der wissenschaftlichen Disziplin des *Software Performance Engineerings* behandelt. In dieser Disziplin gibt es modell- und messgetriebene Methoden welche zumeist isoliert voneinander ablaufen. In der jungen Vergangenheit gab es eine Vielzahl an Innovationen im Bereich der Softwareentwicklung. Hierfür sind im Speziellen die Konzepte der *agilen Entwicklung*, des *Continuous Deliveries* sowie die *DevOps*-Kultur von Interesse.
Die vorliegende Arbeit präsentiert ein neues Vorgehensmodell in der SPE-Domäne, den *ASPE*-Ansatz (***A**gile **S**oftware **P**erformance **E**ngineering*) welcher modell- und messgetriebene Techniken verbindet. Der *ASPE*-Ansatz wurde auf Grundlage des *Travelistr*-Systems evaluiert. Hierbei galt es zunächst, *Travelistr* unter Verwendung des *ASPE*-Ansatzes zu entwickeln und die Nützlichkeit dieses Ansatzes zu evaluieren. Darauffolgend konnte in einem empirischen Versuch mit 32 Testnutzern die Markov-Eigenschaft einer typischen Web 2.0 Applikation bestätigt, und die Effektivität der Markov-Approximation gezeigt werden. Des Weiteren wurden zwei Tools entwickelt die Teilbereiche des *ASPE*-Ansatzes automatiseren: Der *OperationsAndTraceMonitor* und das *UserTrace2Markov*-Tool. Um die Performanz eines Softwaresystems anhand der Warteschlangetheorie evaluieren zu können, wurde ein Ansatz zur Berechnung von Ankunftsintervallen an Stationen auf Basis der Ergodizitätstheorie und Little's Gesetz entwickelt.

Abstract

Performance, considered in this work with respect to *timely* responses, is a key-characteristic of software systems. There are multiple performance engineering approaches in scientific literature. However, most of these approaches lack utility due to the high overhead and additional complexity. In practice, performance management of a software system is often more of an ad-hoc trial and error approach, rather than an engineering approach. Managing the performance of a software system belongs to the field of Software Performance Engineering (SPE) where two distinct approaches are available: The model-based approach and the measurement-based approach. Both having different limitations which may be mitigated by combining them. In the last decade, a wide range of innovations in the software engineering domain emerged. Of interest for this work are especially the concepts *Agile development*, *Continuous Delivery* and the *DevOps*-culture. Besides other benefits, these concepts enable better performance management.
This work introduces a new engineering approach to the SPE domain which showed utility in practice: The *ASPE*-approach (***A****gile* ***S****oftware* ***P****erformance* ***E****ngineering*) that integrates and combines model-based and measurement-based techniques throughout designing and developing a software system. In order to be useful in practice, it is essential to have high utility while bringing little overhead. Therefore, the *ASPE*-approach is built upon the principles (1) *Force automation*, (2) *Reuse artefacts* and (3) *Use approximations instead of over-engineering SPE*. Principles 1 and 2 are tackled by the use of model-driven concepts. Principle 3 is tackled by the use of Markov-models for describing user behaviour and in consequence workloads on resources.
The *ASPE*-approach was evaluated in a case study covering the development of the Travelistr software system which was later also used for an empirical user test where the Markov assumption for a typical Web 2.0 application was evaluated and showed effectiveness. Two self-developed tools that automate parts of the *ASPE*-approach are introduced: (1) The *OperationsAndTraceMonitor* - an instrumentation tool able to track user navigation and the duration of defined operations. (2) The *UserTrace2Markov*-tool - which uses the results of the *OperationsAndTraceMonitor* and semi-automatically calculates Markov model transition probabilities of first and second order. Furthermore, for analysing a software systems performance with the widely used queuing network formalism, an approach to transfer Markov models towards arrival-rates of resources based on ergodicity theory and Little's Law is introduced.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Overview **1**
1.1 Introduction and Motivation 1
1.2 Problem statement 2
1.3 Aim of the work 3
1.4 Methodological approach 4
1.5 Motivating example 4
1.6 Structure of the thesis 9

2 Agile software engineering, continuous delivery and the DevOps culture **11**
2.1 Overview 11
2.2 Software engineering fundamentals 11
2.3 The agile approach 12
2.4 The DevOps culture 13
2.5 The need for automation 14

3 Software Performance Engineering, Markov models and queuing networks **15**
3.1 Overview 15
3.2 Software Performance Engineering 16
3.3 Stochastic processes 17
3.4 Markov models 20
3.5 Queuing theory 22

4 Cloud computing and model-driven engineering **25**
4.1 Overview 25
4.2 Cloud computing 25
4.3 Model-driven engineering 26

4.4 OASIS TOSCA . . . 27

5 Related work **29**
5.1 Overview . . . 29
5.2 Intermediate formats . . . 30
5.3 Performance annotations . . . 35
5.4 Transformation approaches . . . 36
5.5 Service demands and performance measurements . . . 38
5.6 Markov models and queuing networks . . . 39
5.7 DevOps and QA . . . 40
5.8 Other related work . . . 41

6 An approach for Agile Performance Engineering **43**
6.1 Overview . . . 43
6.2 Information requirements and basic elements . . . 45
6.3 The CETO- and MUPOM metamodels . . . 48
6.4 The ASPE-process . . . 50
6.5 Profiling . . . 52
6.6 Model transformations . . . 56
6.7 From Markov models to queuing networks . . . 59

7 Evaluation **61**
7.1 Overview . . . 61
7.2 Objectives of the case study . . . 62
7.3 Data collection and Tools . . . 62
7.4 Travelistr . . . 63
7.5 Sprint 0 . . . 63
7.6 Sprint 1 . . . 68
7.7 Sprint 2 . . . 73
7.8 Sprint 3 . . . 76
7.9 Benefits and Overhead . . . 82
7.10 Threads to validity . . . 82

8 Conclusion and Outlook **85**
8.1 Future research directions . . . 86

Acronyms **89**

Results of the empirical user test **91**

List of Figures **105**

List of Tables **106**

Bibliography **107**

CHAPTER 1

Overview

This chapters purpose is to give an overview and a clear understanding of what the problem domain is and why this domain is interesting for research work.
First, a short introduction about the involved concepts, techniques and approaches is given with special focus on their combinations. Then the problem domain and the goals of this thesis are outlined. At the end, the methodological approach, describing in what way the problems are tackled, is presented and finally the structure of this thesis is outlined.

1.1 Introduction and Motivation

Performance is an important non-functional requirement and a key-characteristic of software systems. Performance is considered here with respect to *timely* responses of software components and entire software systems.
It is useful to take performance already in early phases while designing the architecture of a system into account [84]. Not only is it important to know if a software system can meet certain performance constraints, but also with what hardware and furthermore at what cost this can be achieved. It is essential for good software to maintain good performance in an efficient way.
In practice, performance management is very often more an ad-hoc try and error- than an engineering approach [46]. Managing the performance of a software system belongs to the field of software performance engineering (SPE) [85].
With the rise of agile software development methodologies, the importance of continuous delivery and thus the early provisioning of operations resources increased. As of today, the DevOps culture, which highlights the high interdependence and need for collaboration of development and operations, has gathered relevance in the last years [67] and is widely accepted and practiced today [1]. When combining the agile methodology with the DevOps culture, performance issues are more likely to already be unveiled in early stages

of development due to fast and incremental deliveries. However, there is still a strong need for adequate performance management.
There are two main approaches to analysing the performance of software systems: The measurement-based approach and the model-based approach [85]. Both having different limitations that may be mitigated by each other. Given the new possibilities that agile development, continuous delivery, DevOps and cloud solutions offer, it appears practically usable to combine the model-based approach with the measurement-based approach resulting in better performance engineering.
In the design phase of a software system predictive knowledge can be used to analyse a software systems expected performance constraints to some extent. Once parts of a software system during development are deployed, it is possible to gather empirical knowledge through profiling-observations and enrich the model.
Markov models as well as queuing networks have shown their usability in many different fields, and they are also used to describe the performance characteristics of software systems (e.g.: [32, 46, 47]). Performance models in form of queuing networks for software systems are useful because the performance of a software system is foremost affected by shared resources and their contention leading to queues and in consequence waiting times [70].

1.2 Problem statement

Software performance engineering (SPE), alongside a software engineering process, covers any activity related to performance management of software systems [85]. It aids in creating software systems that satisfy performance requirements, such as Service Level Agreements (SLAs), while efficiently using resources. Furthermore, SPE highlights the importance of engineering principles instead of ad-hoc try and error approaches towards performance management. However, performance management is still split into two areas: The model-based- and the measurement-based approach. In order to be relevant for practitioners, I assume that automated tool-support, that minimizes the overhead of performance engineering, must be available. For comprehensive tool support and reusability, it is essential to introduce standardization especially in the nomenclature, the concepts and the model-formats. With respect to model-driven engineering, standardized metamodels are needed and respective software- and performance model transformation methodologies necessary. All SPE-activities should be seamlessly integrated into a software engineering process without making it unusable for practitioners due to a high overhead. Markov models for describing user-behaviour and in consequence workloads of resources are also used in research. However, an evaluation of the validity of the Markov-assumption given a software systems specific user-behaviour is an essential pre-condition.

1.3 Aim of the work

This thesis will focus on SPE alongside an agile software engineering process.

The research interest and main goal of this thesis is to combine the model-based approach with the measurement-based approach in SPE. Currently, there is to the best of my knowledge no scientific work on a systematic combination of the model- and the measurement-based approach within a software engineering process available.
The performance-modelling approach I will introduce should be useful for analysis and prediction and can be used for simulations, bottleneck analysis, capacity planning and what-if analysis amongst other possible usages.

SG1: The first sub-goal of this thesis is to propose a metamodel that defines all relevant elements and aspects sufficient for analysing the performance of a software system. The metamodel must be capable of integrating both predictive- and measurement data. This thesis will therefore contain an analysis of information requirements for performance models (SG1.1). It is essential to analyse the rich literature in the SPE-domain to outline an appropriate approach that builds upon other research work in this area.

SG2: The second sub-goal is the proposition of an agile software engineering process that includes SPE in a seamless manner. The agile approach must combine the model-based and the measurement-based approach and should contain performance-measure feedback loops throughout. It is essential to evaluate ways to integrate performance objectives, predictions and measures in the software engineering process (SG2.1). Furthermore, I will outline needed model-transformations of this processes in order to be usable in practice (SG2.3). Additionally, the utility of the proposed approach will be evaluated and the needed tools and transformations will be outlined (SG2.4).

SG3: As user-behaviour is described with Markov models in this work, it is essential to outline an approach to derive resource utilization from these Markov models. Especially, an approach to retrieve queuing models from Markov models will be be proposed.

SG4: The fourth sub-goal is the proposition of a metamodel for performance analysis that describes workloads on resources. This model can then be used to simulate a system, detect bottlenecks, what-if analysis amongst many other possible usages. The metamodel must be designed with respect to the stochastic nature of workloads using the Markov model formalism. Therefore, the usability of Markov models in the field of SPE must be analysed (SG3.1). Furthermore, a model-transformation methodology between the metamodel in SG1 and SG3 must be described (SG3.2).

SG5: For modelling the user-behaviour, the Markov-model notation will be used. This also implies that the underlying stochastic process of the user-behaviour fulfils the Markov-assumption. First, a way to determine if the Markov-assumption for a specific system holds has to be defined (SG4.1). Second, an evaluation of the Markov-assumption for a specific web-software system will be carried out (SG4.2).

1.4 Methodological approach

The methodological approach consists of the following steps.

Literature review. I will perform a systematic literature review. I will use different online databases, namely: IEEExplore, ACM Digital Library, CatalogPlus, the online library of the Vienna University of Technology, Google Scholar, dblp: the computer science bibliography of the University Trier as well as the Mendeley research library. I will use primary- as well as secondary studies. For secondary studies I will however also review the primary studies. The research questions and research goals will be answered partly by outlining the systematic review. As we are in the field of software engineering, the systematic literature review will be carried out with regard to the experiences of Kitchenham et al. in the software engineering domain [35].

Proposition of an agile software performance engineering workflow and two metamodels. For the elaboration of my contributions I will apply the principles and use the guidelines of Hevner et al. [81] in order to design a model as an artefact, that improves the process of agile software engineering with respect to performance management in an iterative manner including reviews and evaluations of the artefacts.

Evaluation of the propositions. The evaluation will be carried out using a case study. I will follow the recommended practices for software engineering case studies that Höst et al. [69] presented. The case-study will be of both explanatory and exploratory nature.

1.5 Motivating example

Travelistr is an application designed for travellers. It enables people to share pictures of their journey with other travellers that are nearby. Users are able to share pictures with the Travelistr-App that is a web-application which can be used through a browser on any device. The pictures are enriched with information about the geographic location where the picture was made. Users have the possibility to like pictures, but always only one out of three pictures that are presented to them. The algorithm that selects the three pictures showed take different aspects into account. These aspects are the geo-location and distance to the user. Furthermore it takes the »*impact*«into account. The impact of a picture is determined by the number of *Likes* it has and the date it was taken, whereas recently published pictures are favoured.

Following, three Travelistr software models are depicted. A Use-Case diagram in figure 1.1, a Sequence diagram for the »*Publish Picture*«-Use-case in figure 1.2 and the Component-diagram in figure 1.3 including the nodes where the components are deployed.
The Travelistr software system is used throughout this thesis for evaluation of the proposed concepts as well as to clarify arguments and highlight concepts. In Chapter 7, a case study based on the Travelistr software system is described.

At an early phase in a software project, one might ask questions towards performance of the planned software system such as: *Given these classical software models, how can the*

performance quality of the software system be predicted? What improvements shall be done in architecture, in order to perform more efficiently? Do we need more computing resources? Shall a load-balancer be added for the ImageScaler-/ImageStore-/Database-component?
How can performance constraints (such as SLAs or QoS constraints) be defined in a model-theoretic way? How can i check if the software system will satisfy these constraints?
How can software models be transformed to Performance models that can be evaluated with different approaches and tools?
How can a-posteriori knowledge about the system and the behaviour of the system, be integrated during development and after release? How much overhead is software performance engineering? Are there automated tools available?

This theses aims to propose an approach and describe results of a case study, that tackle these and similar questions.

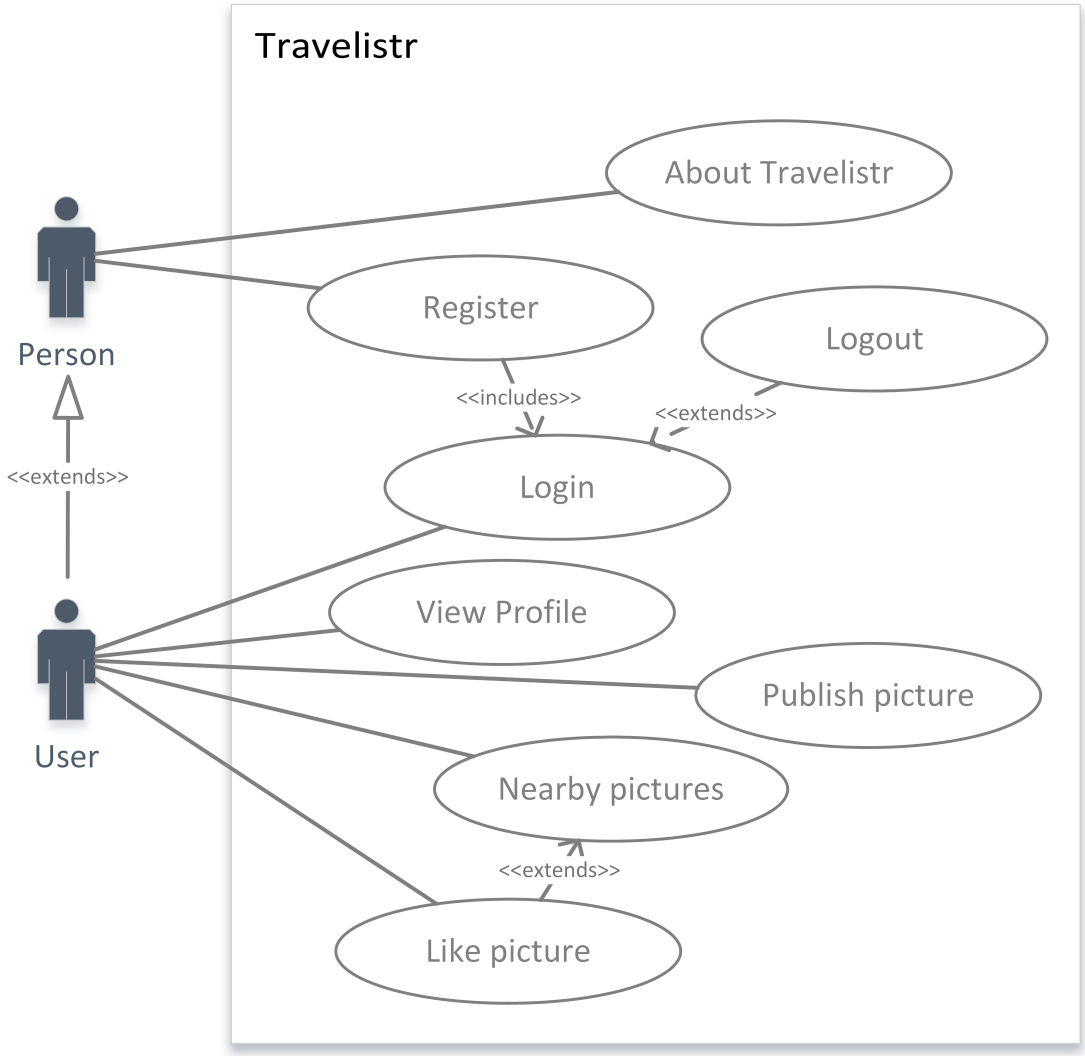

Figure 1.1: UML Use-Case-diagram of the Travelistr software system.

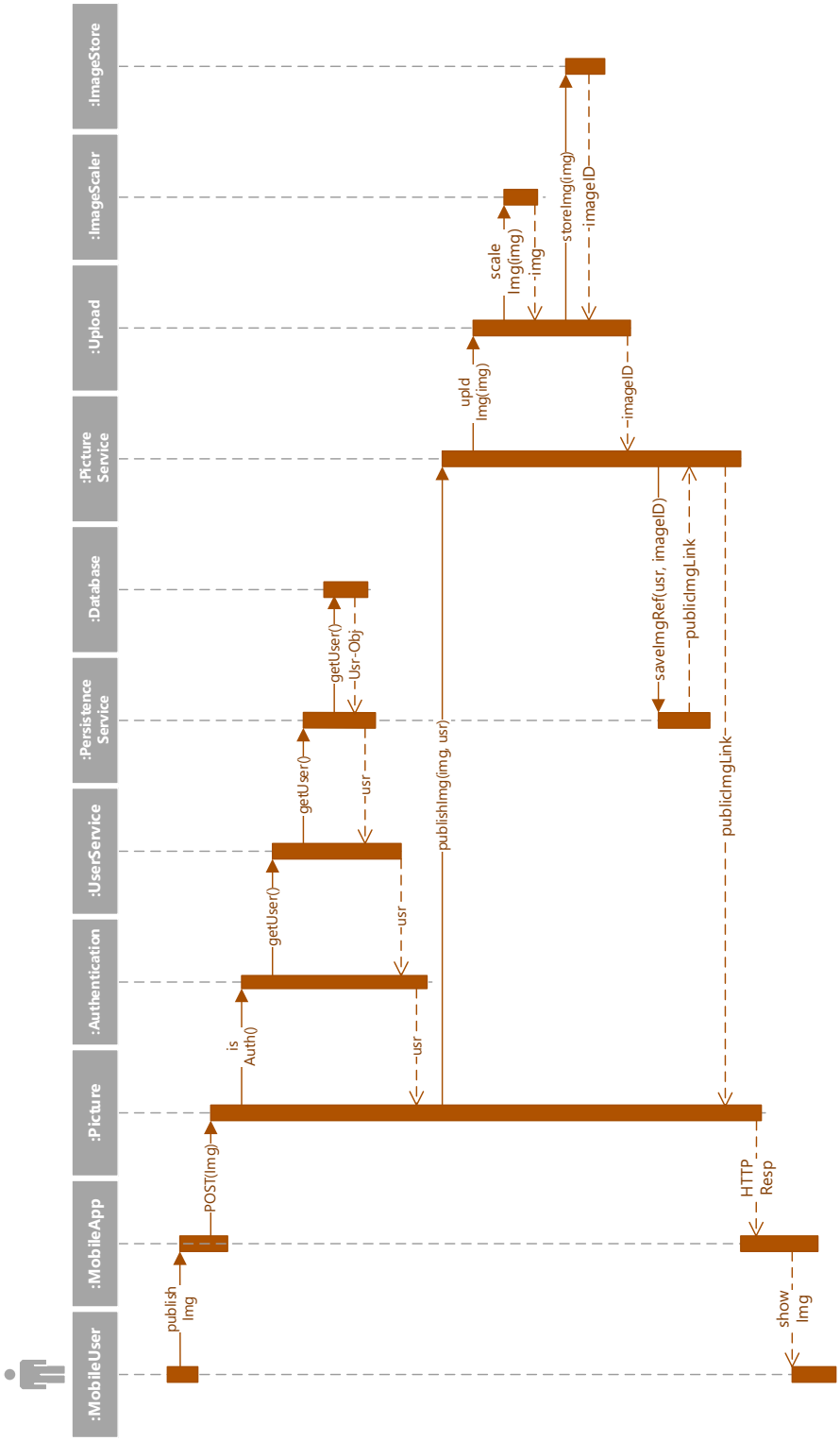

Figure 1.2: Sample UML Sequence-diagram for the Use-Case *Publish picture* of the Travelistr software system (Error cases are left out for simplification).

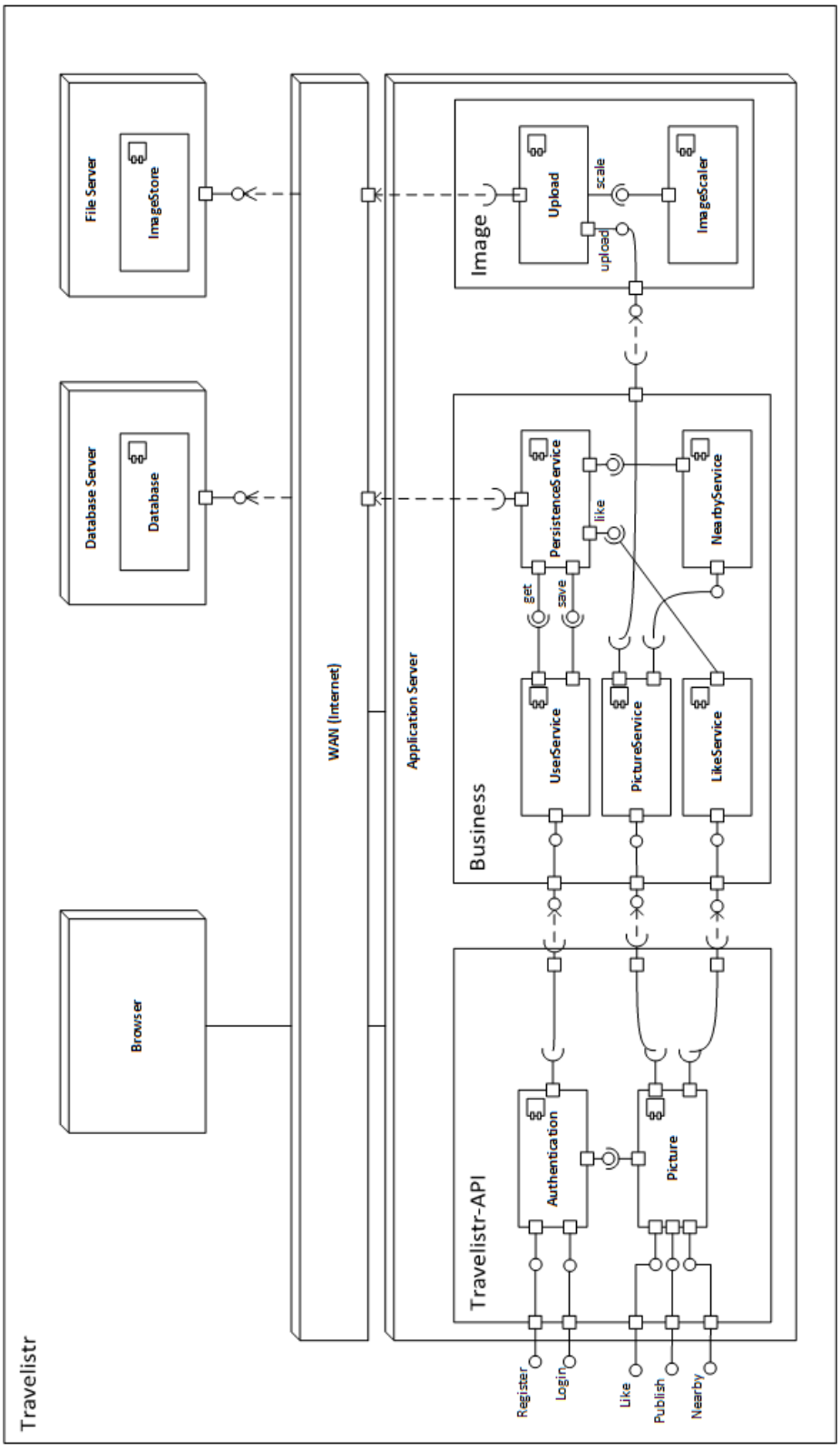

Figure 1.3: UML Component-diagram of the Travelistr software system.

1.6 Structure of the thesis

The theoretical background including the used terminlogy and mathematical foundations will be covered in the Chapters 2, 3 and 4.
Chapter 5 summarizes relevant research in the problem domain and together with the theoretical background constitutes the knowledge base that is used in chapter 6.
In Chapter 6, a new approach towards software performance engineering integrated in agile software development is outlined.
The hypotheses propositions of Chapter 6 will be evaluated in Chapter 7 within a case study. Finally in Chapter 8, the findings and results are summarized and additionally future research directions and pending issues are outlined.

CHAPTER 2

Agile software engineering, continuous delivery and the DevOps culture

2.1 Overview

After outlining the aim of this thesis in the previous Chapter, this Chapter gives an overview on the three concepts: *agile development*, *continuous delivery* and *DevOps*. These concepts emerged in the last decade and are important in software engineering practice today [1]. Model-driven engineering and performance engineering will only be covered briefly here. For more details see the respective Chapters 3 and 4.

2.2 Software engineering fundamentals

»*Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use*« [77]. This definition contains two main propositions: First, the creation of software is an engineering approach. Second, it is concerned with »*all aspects*« that come along the software creation process, not only technical aspects.

A key-characteristic of engineering disciplines is to reuse components and best-practices that showed efficacy in the past. Therefore it is essential that software is developed in a reusable and component-based way. The components have clear interfaces and can therefore be separated from other components. Components can be replaced and exchanged. A software-component offers a certain functionality, can be accessed by a defined interface and may have dependencies. More information on component-based software is given in Chapter 4.

Creating software with whatever kind of software development methodology consists of different phases. The following phases are the most essential and may occur multiple times and in different orders depending on the used methodology.

- **Requirements** from a user perspective
- **Specification** from an engineering view
- **Project planning** from a project management view
- **Design** from a designers/technical view
- **Implementation** from a technical view
- **Testing**
- **Integration** - Binding components together from a technical view
- **Operation and Maintenance**
- **Retirement**

Release engineering is a software engineeering discipline and its goal is to establish and improve a software deployment process in a reliable and predictable manner [40]. **Continuous integration** is a practice where changes are integrated to a main line of development continuously. Developers therefore use a central repository where changes are pushed to and integrated. The term integration corresponds to any level of granularity, components as well as finer or more coarse grained deliverables. **Continuous deployment** is a practice deploying software (or parts of it) continuously to a production or some sort of production environment. In this sense, **continuous delivery** is a term that describes the ability to continuously deliver a software product to customers or any other stakeholder.

2.3 The agile approach

In the last decade, the agile software engineering methodology became increasingly important. The agile approach is especially well suited for projects with possibly changing requirements. Core-concepts of the agile approach are:

- **Incremental development** in
- **short iterations** with
- **fast and continuous deliveries** of running software.

Due to these concepts, a broad range of knowledge in different fields is needed early and in parallel in the software creation process. For example, the deployment of running software after the first software development iteration triggers the need for a deployment environment, thus operations infrastructure is needed already at this early stage. Another consequence of the agile approach consequently is the increased demand of tool support especially for automated processes. SCRUM is a prominent example for an agile software development framework.

The agile approach changed the way, software is created. As a consequence of that, the DevOps culture gathered increasing relevance in the last years [1].

2.4 The DevOps culture

DevOps is a term that stresses the strong interdependence between development and operations. DevOps is a culture and movement that incorporates many different areas inside an organisation such as sharing knowledge and ideas, working together as one team instead of separation, automating workflows and routine tasks amongst others [49].

There are two interpretations of DevOps used in industry: *DevOps as a culture* and *DevOps as a job description* [67]. The term DevOps is used in this thesis to describe a culture where development and operations converge. Furthermore, DevOps is not limited to only development and operations teams following the definition of Dyck et al. [40]: »*DevOps is an organizational approach that stresses empathy and cross-functional collaboration within and between teams - especially development and IT operations - in software development organizations, in order to operate resilient systems and accelerate delivery of changes*«.

In classic waterfall-models for software development, the development-phase was separated from Quality Assurance (QA) and operations. The operations- and QA teams took over after the development team finished their work. [67]
A meme called »*Disaster Girl*«that i think points out a problem of this seperation in a funny way can be found in [7]. The text at this Meme says: »*Worked fine in dev. Ops problem now.*«

DevOps is a cultural shift in organizations that highlights collaboration, communication and interaction between teams, especially the development team with the operations team. Another aspect of DevOps is its focus on Quality Assurance. In the minimum realization of the DevOps culture from a development-perspective it covers the phases *Implementation*, *Testing*, *Integration* as well as *Operation* and *Maintenance*. Collaboration between development and operations results in more realistic testing and better feedback for development. In the same sense, software performance management can benefit from better collaboration.

2.5 The need for automation

Humble et al. [48] argue that in order to continuously deliver software, the deployment process must be highly automated.

In classic waterfall-models for software development, software artefacts are, in the best case, deployed exactly once at the end of the development phase. In the last decade, the situation changed. As a consequence of the the agile methodology, the DevOps culture and continuous delivery, tasks like the deployment of artefacts to an operation environment became routine tasks. Deployments are done much more often. In extreme cases after each source-code-commit, but at least after each iteration. As the effort of manual deployment of artefacts might be reasonable if it is done only a few times in the end of a project, on a regular basis automated tool support is crucial towards cost and performance of development.

Given these circumstances it is not surprising that in the last years the demand for automated solutions and tools increased and many new products came to market, open source as well as proprietary ones. Continuous delivery is essential to agile software engineering and alongside with that, a deployment/operational environment has to be set up, configured and orchestrated. The DevOps community actively supports deployment automation by sharing knowledge and reusable artefacts. The artefacts are mainly of the field of *Configuration management* and *infrastructure as code* in the cloud-environment. [83]

CHAPTER 3

Software Performance Engineering, Markov models and queuing networks

3.1 Overview

In Chapter 2, the agile methodology and its impact on processes and cultures was described. The goal of this chapter is to introduce the concepts of Software Performance Engineering (SPE). Furthermore, Markov models and queuing networks are described and the mathematical foundations for the Chapters 6 and 7 are laid here. The queuing network formalism is widely used in SPE (e.g. [32, 46, 47]). Additionally, hidden Markov models (HMM) are also described. Their use in SPE is rather new [41].

Performance is an important non-functional requirement and a key-characteristic of software systems. The performance quality describes the degree to which a software system or a software-component meets its performance requirements. In this work, performance quality is considered with respect to *timeliness*. Consequently, the performance quality is quantified with time intervals for responses. *Responsiveness* can be quantified within a system as interaction-times between components. Responsiveness can also be measured by interaction-times between users and a software system. The *scalability* of a system describes its capability of still meeting performance quality criteria given an increased workload-intensity. *Throughput* is defined in this thesis as the maximum number of operations that can be processed by a software system in a specific time interval. Throughput is defined here as the maximum of a servers-capacity and a given workload [58]. *Trashing* is a term that describes the phenomena of decreasing throughput when the workload exceeds a certain point [58].

3.2 Software Performance Engineering

»*Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements*« [85].

SPE is a software engineering discipline. In software engineering, different graphical- or formal models may be used. In this context, it is necassary to distinguish between two types of models: (1) ***software models*** and (2) ***performance models*** [86]. Software models have their primary focus on software system descriptions, workflows and interactions between entities. Examples for classical software models are sequence-, activity- and deployment diagrams. On the other hand, the focus of performance models are performance evaluations. Performance models can be used for simulations and analytical evaluations in a predictive way.

There are two different approaches to evaluate the performance of software systems, the (1) **model-based approach** which is carried out early in a project and predicts the performance quality of a system and (2) the **measurement-based approach** that is carried out after development when the system is already deployed. There is a need and tendency for both approaches to converge. [85]

This thesis aims at combining the model-based with the measurement-based approach by capturing the stochastic nature of software systems in a model-driven way. There are many examples for stochastic systems. In terms of software systems, *workloads* are the primary stochastic element. The concept of workloads is used in a wide range of literature for both, model-based- and measurement-based approaches (e.g. [23, 62, 79]). A workload is the amount of work that requests some sort of service in a specific time interval. This workload is stochastic in the sense that it changes and occurs in a non-deterministic way. An example of software systems with stochastic workloads are websites. The workload is primarily generated by the stochastic user surfing behaviour on the website.

Markov models may be widely used to model a software systems stochastic user behaviour (e.g. [32,41,46]). On top of that, queuing networks are often used to analyse the utilization of resources (e.g. [32, 46]). Markov model- and queuing network formalisms enable the analysis of a software systems performance quality. Other examples for performance modelling formalisms are layered queuing models [43], Petri nets, stochastic algebra models, simulation models .... In order to get insights about a systems characteristics and solve stochastic models, two distinct approaches are available: *analytical solutions* and *simulations*.

Requirements towards software systems may change multiple times within a software project. This also triggers necessary changes in the SPE-domain. Software models as well as performance models need to change too. Therefore, in the same sense as in continuous delivery (Section 2.5), I assume that also in the SPE-domain, a high degree of automation is essential for being practically usable.

3.3 Stochastic processes

A *stochastic process* is a mathematical tool in the field of probability theory. The term *stochastic* describes its randomness and the term *process* describes the dependence and the evolution of the state over time.

A stochastic process is an indexed collection of random variables that express the timely change of the state of a system. The state of a system at a time-point is represented by the values of the random variables at that moment. The degree of randomness is always correlated to the available information. The more an observer knows about a system, the less randomness is included in the model and the more deterministic the model is.

Stochastic processes can be classified with a range of criteria. Of interest for this thesis is the classification based on the structure of their stochastic dependencies. These dependencies are mostly defined using conditional probabilities. In this sense, stochastic processes can be classified into two groups:

- Markov processes
- Martingales

A Martingale is a model for a stochastic process in which the knowledge about the current state nor the past states does aid in predicting the future. In consequence, Martingales are referred to as models for *fair games*. A prominent example for a fair game is coin tossing. Here the information about the past coin tosses does not aid in predicting the next coin toss result.
A stochastic process is a Markov process if it satisfies the *Markov-property*. The Markov-property can be defined of any order. The Markov-property of order one states that the future state depends only on the present state. The Markov-property of order two states that the future only depends on the present and one previous state. And so forth.
So in conclusion, a stochastic process is Markov, if, with a limited information of the past, it is possible to predict the future exactly as good as if i had the complete information about the past. In Section 3.4 more information about Markov models is given.

In order to explain the concept of random variables and stochastic processes more precisely, it is necessary to begin with the definition of a probability space.

3.3.1 The probability space

A probability space is a mathematical formalism used to model an event which outcome depends on randomness.

The notion of a probability space was introduced by Kolmogorov. A probability space is defined by the triple Ω, Σ and P.

Ω describes the sample space which includes all possible outcomes of the event.
Σ is a set of subsets of Ω.
P denotes the assignment of probabilities to the sets of Σ.

A probability space needs to fulfil three axioms:

1. $\mathrm{A} \in \Sigma; \Pr(A) \rightarrow [0,1]$.
   Every element A that is an element of Σ has a probability between 0 and 1 assigned.
2. $\Pr(\Omega) = 1$.
   The probability of the whole sample space is 1.
3. $A_k \cap A_l = \emptyset, k \neq l : \Pr(\bigcup_{n=1}^{\infty} A_i) = \sum_{n=1}^{\infty} \Pr(A_i)$.
   For any sequence of mutually exclusive events, their joint probability is equal to the sum of the individual probabilities, called $\sigma - additivity$.

3.3.2 Random variables

A random variable X is a measurable function (in terms of *Borel measurability*) that assigns a numerical value in the state space $\mathbf{R}$ to every possible outcome in Ω of a random event. $X : \Omega \longrightarrow \mathbf{R}$

3.3.3 Law of total probability and conditional probabilities

The following statement holds for any probability space (Σ, Ω, P):

If the sequence (H_k) for $k = 1, .., N$ is a finite or countably infinite subset of Ω with $\Pr(\bigcup_{k=1}^{N} H_k) = 1$ and $\Pr(H_k) = 1 \quad \forall k = 1, .., N$ then $\forall A \in \Omega$ the following statement holds:

$$\Pr(A) = \sum_{k=1}^{N} \Pr(H_k) \times \Pr(A|H_k) \tag{3.1}$$

The conditional probability for two elements $A, B \in \Omega$ with $Pr(B) \neq 0$ is defined as follows:

$$\Pr(A|B) := \frac{\Pr(A \cap B)}{\Pr(B)} \tag{3.2}$$

3.3.4 Formal definition of a stochastic process

Now that the necessary probability background was outlined, it is time to formally describe a stochastic process.
A stochastic process is a collection of random variables X_t that are indexed by a parameter t where t is part of an index set T. Typically the parameter t represents time. If t is of

type integer, the stochastic process is in discrete time. If t is of type real number, the stochastic process is in continuous time.

A random variable X_t usually depends on earlier values of states of the process. For example X_t will depend on $X_t - 1$,$X_t - 2$, .. in the discrete case, similarly this also holds in the continuous case. This is the time-dependence of stochastic processes.

The probabilistic rules of so-called *stationary processes* are constant over time.

Ergodic systems

A system is ergodic if it is (1) *irreducible*, (2) *aperiodic* and (3) *positive recurrent*.

(1) A system is *irreducible*, if it is possible to reach each state from any other state. The probability of being in state j after n-steps starting from i must therefore be greater than zero (see equation 3.3).

$$\Pr(X_n = j | X_0 = i) > 0 \tag{3.3}$$

If 3.3 also holds in the opposite direction (starting at j and reaching i after n-steps with a probability bigger than zero) the states i and j *communicate* denoted by $j \longleftrightarrow i$. The communication-relation is *symmetric* due to its definition. Furthermore it is *transitive* and every state communicates with itself.
A system is irreducible, if all states communicate with each other.

(2) A system is *aperiodic*, if the system state is not systematically connected to time.

(3) A system is *recurrent* if all states are recurrent. A state is recurrent, if the summed probability of returning to that state for an infinite number of steps n is infinite (equation 3.4).

$$\sum_{n=1}^{\infty} P_i^n i = \infty \tag{3.4}$$

A system is *positive recurrent* if the system periodically restarts itself in finite time and every state is visited infinitely often. The expected return time from a state i to itself must be in finite time.

$$\mathbb{E}(\min n \geq 1 : X_n = i | X_0 = i) < \infty \tag{3.5}$$

To be positive recurrent, a system must be *irreducible* and *aperiodic*.

3.4 Markov models

A Markov model is a stochastic model that fulfils the Markov property. The **Markov-property** in this thesis will be regarded primarily of order one and can be summarized as follows: *The Future is independent of the past, given the present.*
A more formal definition of the Markov-property is given in Equation 3.6:

$$\Pr(X_{t_k}|X_{t_{k-1}}, X_{t_{k-2}}, ..., X_{t_1}) = \Pr(X_{t_k}|X_{t_{k-1}}) \tag{3.6}$$

where t_k denotes a set of times $t_k > t_{k-1} > ...$

All necessary information is encoded in the present state, therefore information about the past is not needed. The Markov-property is also called *memoryless*-property. Examples for Markovian-distributions are the Poisson- and the Exponential distribution.

Stochastic processes with the simplifying Markov-property are usually easier to study and analyse. Many theories are built upon this simplified model. However, for modelling a real world system it is important to first check whether the Markov-property holds.
The term **Markov-assumption** is used for systems in which it is assumed that the Markov-property holds. The simplest case of a Markov model is a Markov chain. Another special case of a Markov model is a Hidden Markov Model.

3.4.1 Markov chains

Markov chains have a discrete state space that can be finite or infinite. Markov chains are either in discrete- or continuous time. Events in discrete-time Markov chains can only occur in fixed points in time whereas events in continuous-time Markov chains can occur at any point in time.
If the state transition probabilities do not change over time, a Markov-chain is *stationary*. In the case of a discrete state space, stationary Markov chain, the transition probabilities between states can simply be encoded in a *transition matrix*. In continuous-time Markov chains a *transition-rate matrix* is used.

Steady state

This part defines the calculation of the steady state for discrete time Markov chains. The continuous-case is omitted here.
Given a transition matrix P of a discrete-time Markov chain, in order to get the n-step transition probability from one state to the other, the transition matrix P is multiplied with itself n-times.
p_{ij}^n denotes the probability of being in state j after n-steps starting from state i.
For large n ($n \to \infty$), the resulting matrix might converge to a certain distribution. This distribution is called the *limiting probability* denoted by π_j (Equation 3.7).

$$\pi_j = \lim_{n\to\infty} P_{ij}^n \tag{3.7}$$

In consequence, the *limiting distribution* is denoted by Π (Equation 3.8) where M denotes the number of states of the Markov chain. If a limiting distribution exists, it does not depend on the starting state. If Π is the one single limiting distribution of a Markov chain, Equation 3.9 must hold.

$$\Pi = (\pi_0, \pi_1, ..., \pi_{M-1}) \text{ where } \sum_{i=0}^{M-1} \pi_i = 1 \tag{3.8}$$

$$\Pi \times P = \Pi \tag{3.9}$$

There are two possible ways to calculate the limiting distribution:

1. Multiplying P with itself till an equilibrium is reached.
   This approach might be reasonable for small M, but can get quite expensive in terms of computing power with many, maybe infinite states.

2. Solving the *stationary equations*.
   The stationary equations can be derived from equation 3.9. In general, they result in $M - 1$ linearly independent equations.

An irreducible Markov chain has a unique stationary distribution if and only if all its states are positive recurrent [72] (page 34, Theorem 54).
The limiting positive distribution is a Markov chains stationary distribution if the Markov chain is ergodic [72] (page 40, Theorem 59).
The limiting distribution is also referred to as the *steady state*.

3.4.2 Hidden Markov Models

A Hidden Markov Model (HMM) is a Markov model where states cannot directly be observed. Instead of observing states, it is possible to observe emissions of states. These emissions are modelled with a states emission probability distribution. A HMM has two information-chains: The *state path* and the *observed sequence*. Only the observed sequence is visible. The underlying state path on the other hand is a Markov-chain that is not visible and thus hidden, therefore the name *Hidden* Markov chain. HMMs are due to the simplifying stationary- as well as the memoryless assumption well suited for analysis. HMMs are full probabilistic models. Therefore, it is possible to use Bayesian probability theory to show the significance of results or optimize the model. As an example, the Viterbi-algorithm can be used to compute the most probable state path given an observed sequence. [42]

Following the notation of HMMs with discrete observations that will be used in this work and is taken from [78]. Please note that instead of **A** as a notation for the state transition matrix **P** is used.

- T : Observation sequence length
- N : Total number of states in the underlying Markov chain
- Q : Set containing all distinct states of the underlying Markov chain
- q_t : A single state, therefore $Q = \{q_1, q_2, ..., q_N\}$.
- M : Number of distinct observation symbols
- $\mathcal{O}$: Observed emission sequence.
- O_j : A single observed emission, therefore $\mathcal{O}_T = \{O_1, O_2, ..., O_T\}$
- $\boldsymbol{P}$: State transition probability matrix of underlying Markov chain.
- $\mathcal{B}$: Observation probability matrix.
- π : initial distribution of underlying Markov chain.
- λ denotes the hidden Markov model and consists of the triple ($\boldsymbol{P}$, $\mathcal{B}$, π).

Similar notations are used in a range of other publications in the field of HMMs (e.g. [41], [65]).

3.5 Queuing theory

Queuing theory is used to mathematically describe the occurrence of waiting lines in systems. Typical examples for such systems are supermarket checkout waiting lines, highway traffic jam occurrence as well as in recent year the studies of waiting lines for computing resources of software systems.
Queuing theory provides a formalism to describe systems in what waiting plays an important role. Waiting lines occur whenever the demand exceeds the service availability. The goals of queuing theory is prediction, description as well as proposing design improvements of systems.
There are two basic elements in a queuing system:
(1) A number of (limited) resources that are capable of executing tasks, called **Servers**.
(2) **Customers** that request tasks handled by servers.

In terms of software systems, a server might be for example an image server or a CPU. Examples for customers in terms of software systems are other systems utilizing components or simply users browsing a website.
Queuing systems have an *arrival pattern* as well as a *service pattern* of customers. The Servers service customers based on a specific *queuing discipline* (e.g. First-Come-First-Served). A queue-system has a *capacity*, that is the total number of customers in the queue. The capacity may be infinite. Furthermore, a queuing system has a finite number of parallel *channels* with Servers. The arrival pattern of customers (which is the *workload*

of a system) is the primary stochastic element. There are stationary and non-stationary arrival pattern. If the probability distribution describing the arrival pattern is time-independent, it is a stationary process, otherwise a time-dependent process is called non-stationary.
Furthermore, also the service pattern is usually modelled in a stochastic way. Therefore, the queue length depends on the *arrival distribution* and the *service distribution*. Arrival- and service distribution are mutually independent. Queuing networks are notated with a quintuple:

1. Arrival distribution (e.g. M, D, GD, ..)
2. Service distribution (e.g. M, D, GD, ..)
3. Parallel servicing channels [1,∞)
4. Capacity [1,∞)
5. Queueing discipline (e.g. FCFS, LCFS, ..)

This is known as Kendall's notation. In literature, mostly only the first three symbols are used. Capacity and Queuing discipline are only stated if it differs from capacity being infinite and the queuing discipline being First-come-first-served.
As an example, the M/D/1 model has a memoryless arrival distribution (such as Exponential- or Poisson distribution), the service pattern is deterministic and there is one service channel in the system.

A *queuing network* (QN) is constituted of a network of queues and thus a set of service-centres, which in terms of software systems are a systems computing resources.
Performance models in form of queuing networks for software systems are useful because the performance of a software system is foremost affected by shared resources and their contention leading to waiting queues and in consequence waiting times [70]. Queuing networks for performance analysis for software systems are widely used in literature and are useful for predicting the performance of a software system (e.g. [46], [59]). A lot of algorithms and approaches for solving queuing networks is based on the **product form**-requirement [25].
In a queuing network in product form, equation 3.10 holds, which demands the independence of the number of jobs at the queues in the queuing network [46].

$$\Pr(n_1, n_2, .., n_k) = \prod_{i=1}^{k} \Pr(n_i) \tag{3.10}$$

An example for a queuing network in product form is the simple case of a Jackson-network. In a Jackson-network, servers have two types of arrivals:

1. External arrivals following a Poisson arrival process.

2. Internal arrivals from other servers that in consequence also follow a Poisson arrival process.

Product-form QNs have been used as a basis for a wide-range of extensions (e.g. [25,46,70]). An example for such an extension are **Multiclass-queuing networks** where routing probabilities from one server to another not only depends on the two servers but also on the class of a request [46]. In terms of computer systems, different classes of requests demand different services of resources. Multiclass QNs were described in [59]. Furthermore the necessary input parameters for a simple multiclass QN were outlined, these are: (1) classes, (2) devices, (3) request arrival rates on classes and (4) service demand of requests of every class on every resource.
However, as outlined in [59], modern software systems often violate conditions of product-form QN models. Product-form QN models are built upon the condition of independent queues as showon in equation 3.10. Examples outlined in [59] for such violations are blocking, simultaneous resource possession, forking amongst others. All these features are however immanent to modern software systems. As an example, for software systems it is common for an operation to use more than one resource in a nested fashion. E.g. for an upload of a picture, CPU for scaling the image as well as the Image-Server is needed. It is common sense nowadays to design- and develop software systems in a layered fashion. Concerns are separated to different layers introducing abstractions in each layer. An example of the layered sequence of calls and replies of resources can be seen in the Travelistr-example in Figure 1.2.
If an operation utilizes more than one resource simultaneously, *extended queuing network* are a suitable formalism introduced by Sauer et al. for »*simultaneous resource possession*« [70] [71].
A formalism for describing simultaneous possession of resources and including blocking, forking and showed applicability in practice was introduced in [43] as the Layered queuing networks formalism (LQN). More information on the LQN-formalism is given in Section 5.2.4.

3.5.1 Little's Law

Little's Law is a very prominent Operational law that applies to queuing theory. It is both powerful an easy. Little's Law states that the expected number of customers L in any ergodic system is equal to the expected arrival rate λ times the expected spent time of a task W.

$$L = \lambda W \tag{3.11}$$

The theorem is astonishing, as the arrival- nor the service distribution or anything else is taken into consideration. It was proofed by John Little in [56].

CHAPTER 4

Cloud computing and model-driven engineering

4.1 Overview

In the previous chapter, the Software Performance Engineering domain was introduced together with its mathematical foundations in form of stochastic models and queuing networks. The aim of this chapter is to give an overview of topics in the area of Cloud computing and model-driven engineering that are relevant for this thesis. Additionally, the model-driven provisioning specification OASIS TOSCA is introduced. TOSCA eases the provisioning process and the management tasks of component-based systems in the Cloud computing domain.

Cloud computing is a term that describes shared infrastructure that are easily accessible and manageable. This is achieved by hiding away parts of the complexity through abstraction from physical computing resources. Cloud computing gathered a lot of attention in the last decade.

Model-driven engineering (MDE) uses models for knowledge representation. It is a concept that introduces an additional layer of abstraction and so hides away parts of the complexity of software systems. The Object Management Group (OMG) implements MDE with its model-driven architecture (MDA) framework. Model-driven provisioning is a term describing a model-driven approach towards application provisioning.

4.2 Cloud computing

»*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal*

management effort or service provider interaction« [57]. The NIST definition of Cloud computing [57] defines three service models and four deployment models. The service models are: *software as a service* (SaaS), *platform as a service* (PaaS), *infrastructure as a service* (IaaS). The deployment models are: *private-*, *community-*, *public-* and *hybrid Cloud*.

The concepts of Cloud computing are not new. However, technical limitations hindered their realisation. Only in the last couple of years convenient large-scale Cloud platforms from different vendors came to market. Examples for such platforms are Amazon web services (AWS) in 2006, Microsoft Azure in 2010 or the IBM Cloud Bluemix in 2014. Due to its novelty, Cloud computing lacks standardization. Different vendors follow different concepts/standards and/or define their own standards.

In the software engineering process, the production-Cloud[1] is not available in every stage of the life-cycle.
Other servers or more cost efficient Clouds are used instead meanwhile. At the end of development, the software system has to be migrated to a production environment.
There are many other examples that underline the importance of Cloud portability and interoperability as well as the need for standardization.
Due to the lack of standardization, the migration from one Cloud platform to another is not always possible and in some situations practically infeasible. The situation of not being able to change a Cloud vendor is called a vendor lock-in.
Portability and interoperability must be defined in a concrete way inside a Cloud domain as it would contain little information in a general way [39]. Following a definition of different dimensions of portability and interoperability that is partly taken from Di Martino et al. [39]. Four dimensions of portability: Data-, component-, service- and application portability. And three dimensions of interoperability: Component-, service- and application interoperability.

Elastic Clouds are new ways to handle scalability and performance issues. Computing resources can be added on-demand quickly.
I assume that practitioners might be mislead to neglect the importance of performance engineering, as resources in a Cloud can easily be extended. However, this may lead to inefficient and costly software systems.

4.3 Model-driven engineering

Model-driven engineering (MDE) is a model-based approach in the software engineering discipline. At the heart of model-driven engineering are models. A model is a representation of a system or parts of it. Models are transformed either to other models (model-to-model transformations) or to text (model-to-text transformations) which is

[1] The term *production-Cloud* is defined here as a Cloud platform where the application will be provisioned when in productive use.

mostly source code in a specific language. With these formalisms, MDE introduces an additional level of abstraction and separates concerns as it is technology- and platform independent.

MDE increases the efficiency and effectiveness in software engineering and is said to have a significant growth in usage in the future. One of the reasons for that is the convergence of business analysis and software development. [34]
As an example, Gartner Inc. foresees a promising future of MDE: »*By 2018, more than half of all B2E mobile apps will be created by enterprise business analysts using codeless tools*« [9]. MDE separates the platform- and technology independent design from the platform and technology dependent implementation. Therefore, MDE supports portability, interoperability and reusability of components, services and applications [39].

Currently there are several research projects working on the combination of MDE with the Cloud computing domain. An example for such a research project is MODAClouds [20]. The aim of the MODAClouds-project is to support developers as well as operators in the development, provisioning and orchestration of Cloud applications.

Meta object facility (MOF) is a concept defined by the Object Management Group (OMG). MOF describes a multi-layer architecture that's aim is to describe a standard that enables portability of models as well as model-transformations.
Model-driven architecture (MDA) [13] is an »*industry-standard architecture*« [19] based on the MDE principles defined by OMG.

4.4 OASIS TOSCA

The deployment and orchestration of applications is a critical aspect in application provisioning in the Cloud [36]. Especially the portability as well as the automated management of applications and their components is of major interest in the enterprise IT today [31]. The functionality of applications in the Cloud is typically provided by multiple heterogeneous and distributed components [31]. Developing or using component-based software that forces modularization and decomposition of services is therefore an important aspect. It enables the migration of services as well as the exchangeability and reusability of application components.
The OASIS topology and orchestration specification for Cloud applications (TOSCA) [21] is a specification in the model-driven provisioning discipline that has the component-based approach at its core. TOSCA is introducing an abstract, flexible, vendor- and technology neutral way of defining application and infrastructure components. This approach results in portable application- and management descriptions as well as interoperable and reusable application-components that can be managed and deployed automatically. [36]

Binz et al. outline three major challenges that are tackled by TOSCA [31]:

1. Automating the application management

2. Portability of applications as well as the portability of application management
3. Reusing interoperable application components

Applications are aggregates of services and components that process, store and use data. TOSCA describes the components/services and their dependencies to other components/services with a topology graph. These TOSCA topology models are *software models*.

4.4.1 The TOSCA modelling language

TOSCAs meta-model is constituted of the following parts:

1. The *topology template* that consists of components and their relationships.
2. The dynamic behaviour and management of components/services/an application can be modelled with any kind of workflow-language and is defined by *Management Plans*.
3. The topology template and the plans together constitute the *Service template*.

4.4.2 TOSCA containers

TOSCA specifies CSAR (Cloud service archives) as its packaging format. CSAR archives are processed by TOSCA containers such as OpenTOSCA and are processed either imperatively or declaratively. The declarative approach is realized by TOSCAs management plans. The imperative approach is carried out by the respective TOSCA-container itself.

4.4.3 TOSCA policies

The TOSCA specification allows to define and manage non-functional behaviour and quality-of-services of components and services through *policies*.

Types of policies are expressed by *Policy Types*. *Node Types* (e.g. a Database Server), can have multiple Policy Types. A *Node Template* is an instance of a Node Type and can declare to expose instances of the type *Policy Template*. A Policy Template is a specific instance of a Policy Type. The Policy Types element defines which properties are allowed and the Policy Templates element defines concrete values for these properties.

4.4.4 OpenTOSCA

OpenTOSCA [22] is an ecosystem of open-source software and tools that aim to help with the provisioning and orchestration of Cloud-application with TOSCA. It has three parts: (1) A modelling tool *winery*, (2) A runtime environment *OpenTOSCA* and (3) a web-based provisioning platform *Vinothek*.

CHAPTER 5

Related work

5.1 Overview

The foundations and theoretic background were outlined in the previous chapters. It is the goal of this Chapter to summarize relevant research work and research directions in the different domains relevant for this thesis. First, research in the field of model-driven Software Performance Engineering is outlined. A schematic workflow containing the main research areas in this field is given in Figure 5.1. These research areas are:

1. ***Intermediate formats*** that are suited between software models and performance models and enable the exchange and reusability of performance information of a system. As an intermediate format, they also mitigate the overhead of the need for many transformation approaches (Section 5.2).

2. ***Performance annotations*** is a way to integrate performance information, requirements, the operational profile amongst others in software engineering artefacts. Research mainly focuses on classical software diagrams in UML notations. Often used are use case diagrams, sequence diagrams and component diagrams (Section 5.3).

3. ***Transformation approaches*** that are available for model-to-model transformations. The research work was filtered with respect to intermediate formats. The transformations I analysed are thus carried out from a software model to an intermediate model (Section 5.4).

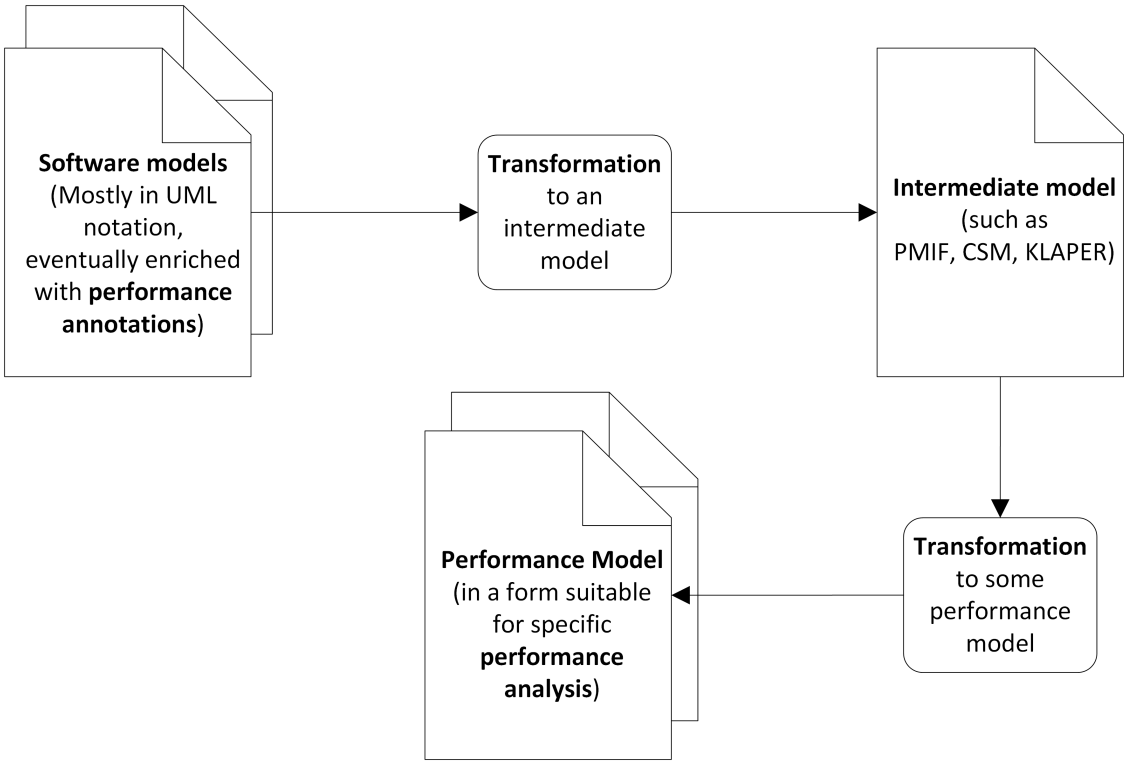

Figure 5.1: The workflow for performance analysis that is used widely in research and is subject of the first part of this literature review (partly based on [86, p. 1530, Fig. 1]).

This thesis aims to combine the model-based and the measurement-based approach for component-based software systems within the DevOps culture using Markov models and queuing networks. Therefore, also research in the field of performance measurement (Section 5.5) and on Markov models as well as queuing networks for performance analysis (Section 5.6) are outlined. Furthermore, work in the field of SPE with DevOps (Section 5.7) is described and additionally, other related research is summarized in Section 5.8.

5.2 Intermediate formats

5.2.1 Core Scenario Model (CSM)

CSM [62] is an intermediate model used in the PUMA framework (Performance through unified model analysis, Section 5.4.1). This intermediate model, similar to any other intermediate format, comes with the purpose of reusability of transformation approaches as well as exchangability of informations. The metamodel of CSM can be found in Figure 5.2.

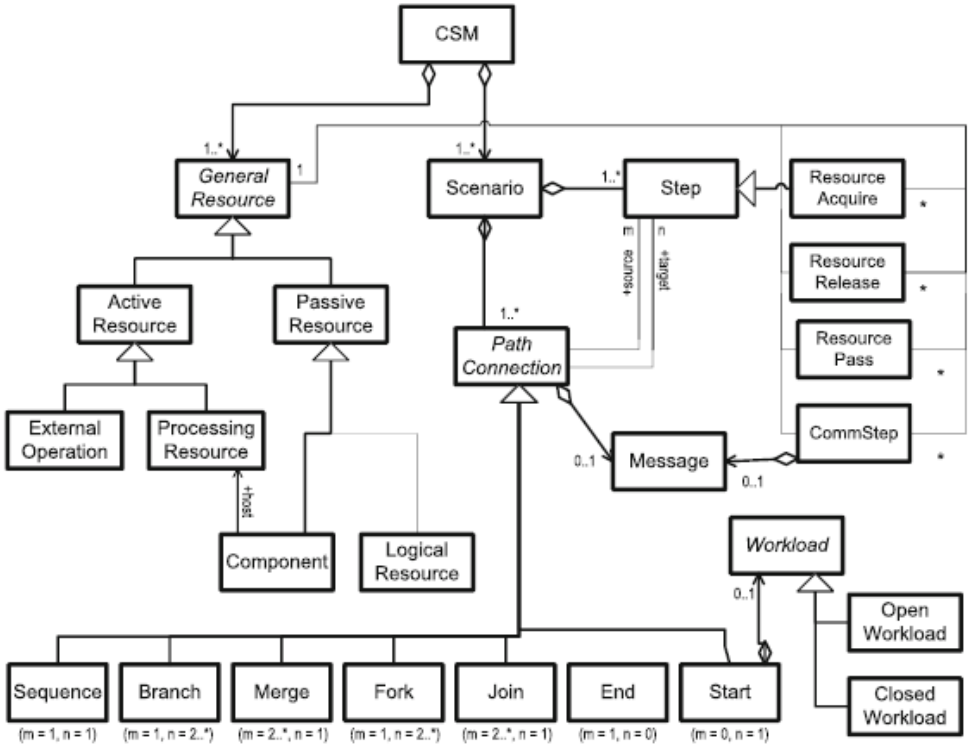

Figure 5.2: Simplified class diagram of the CSM metamodel, taken from [86](p. 1536, Fig. 6).

Following a brief description of some of the CSM model elements.
CSM is based on the same concepts as MARTE (»*UML Profile for Modeling and Analysis of Real-Time Embedded systems*«(MARTE) [23]). It is built around the three elements (1) *Resource*, (2) *Scenario* and (3) *Workload*.
A *Resource* is modelled using the entity *GeneralResource*. A GeneralResource has a scheduling policy and can be either *active* or *passive*, *logical* or *physical*. A *Scenario* is constituted of *Steps*. A *Step* can *acquire* and *release* resources. Two *Steps* are connected with a *PathConnection*. Any *PathConnection* can have a *Message* associated to it which describes the type and size of a message sent from one step to the other. A *PathConnection* defines the relationship and sequence of processing. It can be of the following types:

- *Sequence* for sequential ordering of steps
- *Branch* for an OR split
- *Merge* for an OR join
- *Fork* for an AND split
- *Join* for an AND join

- *End* for the ending of a path and thus a *Scenario*
- *Start* at the beginning of a path and thus a *Scenario*. *Start* also has an association to a *Workload*.

A *Workload* can be *open* or *closed* and defines an *arrival pattern* attribute. The distinction between open and closed workloads is based on the distinction in the field of queuing networks regarding open- and closed networks. An open network has external arrivals- and departures whereas closed networks have a fixed population.

5.2.2 PMIF

The *Performance Model Interchange Format* (PMIF) was first introduced by Smith et al. [73]. Later, the metamodel was revised and introduced again as PMIF2 [76]. PMIF and PMIF2 are built upon the theory of queuing networks.
The authors describe a workflow that uses the concepts of MDE with respective model transformations. Another exchange format they use in this workflow is S-PMIF which is intended to capture architecture-, design- and performance informations about a software system. It is constituted of three different schemas: Topology, ComputerResourceRequirements and Distribution. A translation approach from S-PMIF to PMIF is described in [75]. PMIF2 defines the *QNM metamodel 2* which is a metamodel describing the model elements that can be used in PMIF2 to describe a software system with a queuing network model. The head element of the QNM metamodel 2 is the *QueuingNetworkModel*, it's sub elements are *Arc*, *Node* and *Workload*. Two *Nodes* are connected with an *Arc*. A *Node* can either be a *Server* or a *Non-Server Node*. A *Workload* is linked to a *Server* via a *Service Request*. More detailed descriptions about the PMIF specifications can be found in [14]. The metamodel of PMIF2 can be found in Figure 5.3.

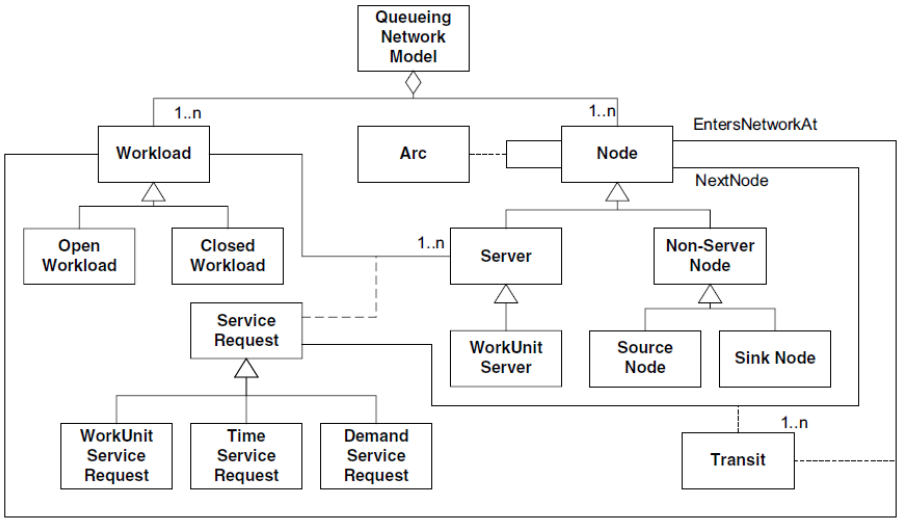

Figure 5.3: Class diagram of the PMIF2 metamodel, taken from [74]

5.2.3 KLAPER

Grassi et al. [45] introduced KLAPER, a *Kernel Language for Performance and Reliability analysis* that is, in its purpose very similar to CSM and PMIF: An intermediate model between software models and performance models. The aim of KLAPER is easing the process of non-functional analysis of software systems. Therefore, the focus is not restricted to performance analysis. In their work, they outline the KLAPER metamodel as well as transformation approaches towards the KLAPER model (Section 5.4.3).
The metamodel of KLAPER can be found in Figure 5.4.

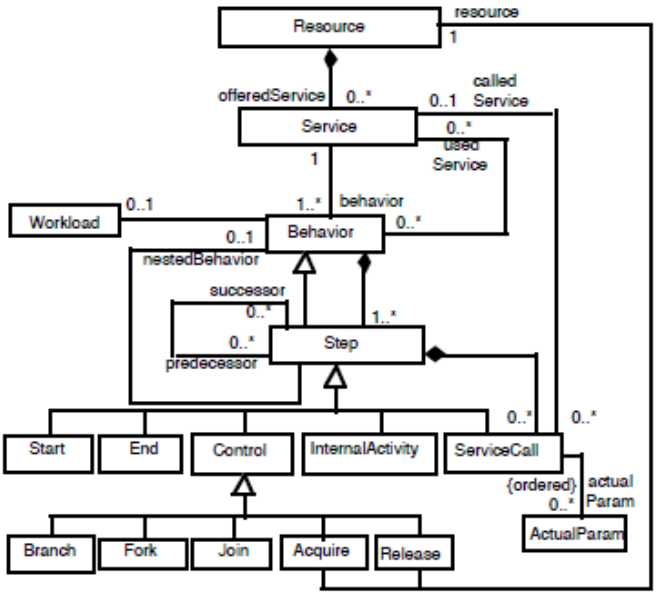

Figure 5.4: Class diagram of the KLAPER metamodel, taken from [45].

5.2.4 Layered queuing networks

Layered queueing networks (LQN) [43] is a notation for extended queuing networks in canonical form. As described earlier, given multiple requests, an operation on a resource may not be serviced at once. Instead the operation gets slices of computing power over time. A canonical form does not take these timely slices into account, instead an operation is modelled as if it was done at once.
Extended queuing networks, and respectively LQN, are able to describe *simultaneous resource possessions* (Section 3.5).
The model elements of LQNs are (1) *Tasks* which represent resources. Tasks run on a physical or logical (2) *Processor* and have a queue. Tasks offer operations defined by their (3) *Entries* and can call operations of other tasks with (4) *Requests*. *Requests* can be of type *Rendezvous* which is a synchronous call that blocks the client till the server replies, *Forwarded* which is an asynchronous call that doesn't block the client and *Send-no-reply* which is also an asynchronous call but with the difference that the client does not get an

answer from the server at any time.
The demand of a service type, and therefore the workload of *Processors* can be defined with (5) *phases* and (6) *activities*.
Phases are specified at the entries of tasks in a numerical way.
Activities are finer grained and are used to model the work steps with a directed *activity graph*. Activities are connected with *Pre-* (*And-Join*, *Quorum-Join*, *Or-Join*) and *Post* (*And-Fork*, *Or-Fork*, *Loop*) operations.
Processors are utilized by activities and are Servers in the sence of classical queuing network models as they only receive requests but do not send requests to others. Processors have a single queue for requests. The execution time of a processor is divided into slices by using the (7) *Group* element. A Group consists of tasks. An example for this is the computing time of a CPU wich is needed by a group of taks. Each task gets some slices of the CPU time to carry out work. [44]
The metamodel of LQN can be found in Figure 5.5.

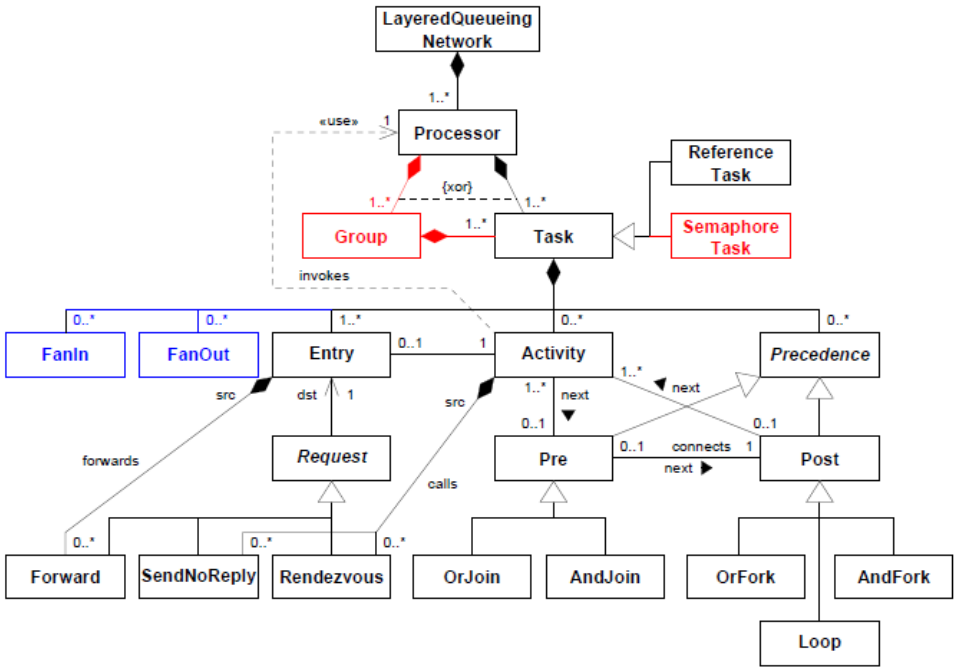

Figure 5.5: Class diagram of the LQN metamodel, taken from [44](p. 3, Fig. 1.2).

5.2.5 Palladio component model

The *Palladio component model* (PCM) [28], is a metamodel used to describe and analyse the quality of component-based software in a model-theoretic way. Its first and foremost goal is to aid in the prediction of performance and reliability. PCM »*explicitly include all factors influencing the performance of a software component*« [28]. These factors are: (1) implementation, (2) external service performance, (3) execution environment

performance and the (4) usage profile. PCM allows to specify quality of service (QoS) in a parametric way and allows the modelling of component behaviour with stochastic distributions. PCM has similar concepts like PMIF, CSM and KLAPER but is novel in that it focuses explicitly on component-based architectures.
One PCM instance is constituted by different models. These models are: (1) *Component specification*, (2) *Architecture model*, (3) *Usage Model* and are specified by different developer roles (Section 5.4.2).

5.3 Performance annotations

5.3.1 UML SPT and MARTE

The »*UML Profile for Modeling and Analysis of Real-Time Embedded systems*« (MARTE) [23] is the successor of UML SPT, the »*UML Profile for Schedulability, Performance and Time*« [24].

The concepts of UML for MARTE are very close to those of the CSM metamodel. Therefore it is only described briefly here. In [86] a more comprehensive explanation to MARTE is given as well as its equivalent model elements in the CSM metamodel (Section 5.2.1). The metamodel of UML SPT can be found in Figure 5.6.

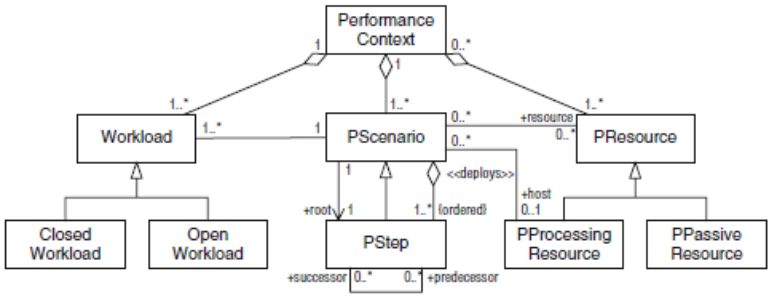

Figure 5.6: Class diagram of the UML SPT performance analysis domain model, taken from [62].

The MARTE profile is constituted of three parts:

- MARTE Foundations
- MARTE Design Model
- MARTE Analysis Model

MARTE foundations are the core of the profile. The *Design model* and the *Analysis Model* extend the foundations. The modelling of real-time- and embedded systems is supported by the *Design Model*. Annotations that are meant to be added to UML models in order

to enable evaluations of the modelled software systems, are defined in the *Analysis Model*. The Analysis Model has support for schedulability (the SAM package) and performance (PAM). PAM (*Performance Analysis Model*) and SAM (*Schedulability Analysis Model*) share the concepts of quantitative analysis in the GQAM package (*Generic Quantitative Analysis Model*). QCAM describes three basic elements: *Resource*, *Scenario* and *Workload*. A *Scenario* is a specific case of user interaction with a system. A *Scenario* is triggered by an event and has one or more *Step*s. A *Step* utilizes a *Resource*. A *Workload* is constituted of one or more events that consequently start a *Scenario*. [29]

5.4 Transformation approaches

5.4.1 The PUMA framework

The PUMA framework (*Performance from Unified Model Analysis*) [87] was first introduced in 2005. This initial PUMA model was built for UML models in combination with UML SPT, the UML profile for Schedulability, Performance and Time.
In 2014 [86], the PUMA framework was adapted in order to use UML MARTE annotations instead of UML SPT. The UML profile MARTE is used to add non-functional properties to software models. For more information see Section 5.3.1.
In their work [86], Woodside et al. present the PUMA framework that uses automation efforts by transforming software models to performance models. Software models have their primary focus on the functionality and architecture of software systems as well as workflows and interactions. Examples for classical software models are sequence-, activity- and, deployment diagrams. On the other hand, the focus of performance models are performance evaluations based on resource usages in the sense of simulations or analytical solutions.
Due to their different focus, software models and performance models contain different information resulting in what the authors call a »*semantic gap*«. For transformations between these two kinds of model, the semantic gap is overcome by using the common elements. In essence, these elements are resources and behaviour. Behaviour is thereby modelled with *steps* that make use of resources. A step therefore is a »*unit of behaviour*«resulting in resource utilization. The main purpose of the PUMA framework is a highly automated creation of performance models based on software models annotated with UML and MARTE (Section 5.3.1).
The PUMA framework is capable of using a variety of software models and performance models. In concrete terms, PUMA allows to transfer software models in the form of interaction-, activity- and deployment diagrams to performance models in the notation of queuing networks, layered queuing networks, Petri nets and simulation models [86]. In [86] furthermore a description of transformation approaches towards LQN and Petri nets are presented.
Resulting performance models can afterwards be analysed or simulated with respective third party tools independent of the PUMA framework. PUMA has the intermediate format CSM (Section 5.2.1) at its core that reduces the efforts for the need of many

transformation approaches.

The PUMA approach defines five steps for transferring software models to performance models:

1. Preliminary: Select the critical usage profile of interest.
   The use cases that seem to be relevant are identified and must be contained in the software models.

2. Add the usage profile information to the software models with MARTE annotations

3. Select and transfer the relevant information of the software models towards a CSM model.

4. Optional: If the performance model will be of type layered queuing network (see Section 5.2.4) an analysis of extended resource properties must be carried out.

5. Transfer CSM to some performance model.

The PUMA framework automates steps 3, 4 and 5. Step 1 and 2 are however manual tasks.

5.4.2 The Palladio component model

Becker et al. [28] introduced the PCM model (Section 5.2.5). In the development of component-based software the authors differentiate four different roles: (1) *component developer* that specifies the components, (2) *System Architect* that specifies the *architecture model*, (3) *system deployer* that specifies the *resource model* and the (4) *domain expert* that specifies the *usage model* and thus the *usage scenarios*, the *workloads* and the *scenario behaviour*.
A PCM instance is built by these different roles and includes different models which are translated in terms of M2M and M2C transformations towards performance models and code implementations.
The authors furthermore provide a tool with a graphical user interface in order to ease the specification of the needed models.

5.4.3 UML and OWL-S to KLAPER

Grassi et al. [45] describe a transformation approach from UML- and OWL-S models to a KLAPER model (Section 5.2.3). Furthermore, they describe a transformation from KLAPER to analysis model, namely discrete time Markov processes (DTMP) and extended queuing networks (EQN).

5.4.4 UML to PMIF2

Cortollessa et al. [37] present a transformation approach from classic UML models to PMIF2 queuing network models. The authors add performance information to software models in UML notation. This additional information is integrated using UML SPT annotations. The authors implemented the transformation by using ATL [3]. The transformation logic is based on the SAP one approach [38].
The outlined methodology in [37] defines a set of transformation rules for generating a multi-chain queuing network model. The software models define static aspects of a software system with component diagrams and dynamic aspects with sequence diagrams, whereas one sequence diagram has to be drawn for every use case.

5.4.5 AutomationML and PMIF

Berardinelli et al. [26] show the combined use of AutomationML and PMIF. AutomationML is a standard for representing and exchanging artefacts between engineering tools. With the combination of AutomationML and PMIF they outlined a step towards early performance validation of CPPS (cyber physical production systems). The authors describe three integration styles between AML and PMIF (namely *native*, *linking*, *transformations*). This study is built upon a case study from Folmer et al. on the evaluation in industrial plant automation [18].

5.5 Service demands and performance measurements

In order to estimate parameters of performance models, direct measurement techniques are available that need performance probes of different layers. E.g. resource-, application-level or request-based. An approach introduced in [53] uses high-level request based measurements to estimate service resource consumptions. Therefore, two approaches are proposed: (1) a linear regression method and (2) a maximum likelihood function. Their approach has several advantages compared to lower-level approaches. For example, request-measurements are easy to obtain, integrating external services is no problem, there is no need to profile a system which may lead in observation overhead that may distort measurements is needed and also factors such as latency are taken into account. The results of an experimental validation show that their proposed approach is effective and often more effective than other approaches based on linear regression using lower-level measurements.

Magpie is a toolchain described in [27] that, based on hardware-, middleware- or application-component level traces that are then correlated to the requests, extract the workload of a software system. The computing demand for any request is then calculated and dependencies are causally ordered. The results can be used as a basis for performance models.

Kieker [68] is a framework for continuous monitoring and analysing the runtime behaviour of software systems. Kieker offers two main benefits: (1) Monitoring of an applications

performance and (2) Tracing of the interactions of components [68]. Based on that, it is possible to execute dynamic analysis of software systems [80]. Kieker is an extendable, open-source project hosted at sourceforge [18]. Researchers as well as companies contribute to it [68]. Kieker allows system-level- (e.g. CPU utilization) as well as application-level (e.g. response times of components) monitoring [11]. Kieker is split in two main parts: (1) *Kieker.Tpmon* and (2) *Kieker.Tpan* [68]. *Kieker.Tpmon* is the component responsible for monitoring and logging data. *Kieker.Tpan* is the analysis-component where the runtime behaviour logged in the monitoring data is reverse-engineered and visualized. Visualizations used by Kieker are for example sequence diagrams and class diagrams. For the purpose of application-level monitoring, components must be annotated at a language-level. For example, in Java this is realised by Java annotations. For the purpose of system-level monitoring, Kieker makes use of the Sigar API [15].

5.6 Markov models and queuing networks

Markov models and queuing networks are two formalisms widely used in SPE (e.g. [32, 46, 47]). Hidden Markov models have been used in many different fields [41]. The first major application of HMMs was in speech recognition [65]. However, the use of hidden Markov models in the field of performance evaluations of software and thus in the field of Software Performance Engineering is relatively new [41].

In [30], the Java Modelling Tools are presented. A tool dedicated to evaluate computer systems performance with queuing models. It is licensed under GNU GPL and consists of several tools such as JMVA, a tool to carry out the Mean Value Analysis for product-form queuing networks.

Hoorn et al. [79] describe an approach for generating probabilistic workloads for web-based systems that are able to vary in intensity. The user behaviour is described by means of Markov models and captured in a *User behaviour model*. Besides that, the authors define three other models: the *Application Model* that specifies the possible sequences of usages with a hierarchical finite state machine, a *User Behaviour Mix* model that defines which user behaviours are used during generating a workload and a *Workload intensity* model that specifies the varying numbers of users over time. Based on this approach, the authors furthermore extended the workload generator JMeter to what they call Markov4JMeter.

Zhou et al. [88] proposed a method for late-cycle website performance analysis using Markov models. They used log files of the user navigation and from there extracted an extended state diagram. Based on the extended state diagram the authors derived a user behaviour model by using the Markov model formalism. This model was then used for performance analysis.

Jespersen et al. [50] evaluated the Markov assumption for the mining of web usage. The authors examined the quality of rules derived from two websites with the Hypertext Probabilistic Grammar model (HPG) [33]. HPG is a method that relies on the Markov

assumption where the probability of the next state is only determined by the current state. HPG is a method for websites, taking all pages as states and the hyperlinks between pages as transitions.
For the quality evaluation they defined two measures: (1) similarity and (2) accuracy. Similarity compares the amount of rules that were derived with HPG that are equal to the true usage patterns. Accuracy compared the derived probabilities of the rules with the true usage patterns. As a result, the authors suggested that Markov based approaches are better suited for tasks that need less accuracy.

Li et al. [55] propose an easy approach for evaluating the Markov property for modelling user behaviour. In this approximative approach, the transition probabilities only taking the current state into account are compared to the transition probabilities when taking the current and one previous state into account. This approximative approach was demonstrated in a case study of the website `www.seas.smu.edu` for web usage profiling. A definitive answer if web usage can be modelled with Markov chains is not given however as the empiric case study has several limitations.

Rabiner et al. [66] describes three basic problems of HMMs that must be tackled in order to be useful in practice. In addition Muntz et al. [41] added a fourth problem. The authors also show the solutions to these problems in their work ([66] as well as [41]).

HMM - Problem 1

Given a model λ, what is the probability of observing the sequence of emissions $\mathcal{O}_T$: $\Pr(\mathcal{O}_T|\lambda)$?

HMM - Problem 2

Given an observation sequence $\mathcal{O}_T$, find an optimal state sequence in a specific sense.

HMM - Problem 3

Construct a model λ given the observations $\mathcal{O}_T$ where $\Pr(\mathcal{O}_T|\lambda)$ is maximized.

HMM - Problem 4

Given a set of Models λ_j as well as observations $\mathcal{O}$: With what probability is λ_j the actual underlying model.

5.7 DevOps and QA

Roche [67] describes the transition of classic quality assurance (QA) after development in the waterfall model towards a more mature software engineering process that integrates QA throughout design and development by adopting DevOps practices. DevOps alongside an agile software development process and QA therefore converge. Feedback mechanisms

aid in making informed decisions by using large datasets of quantified metrics. Unlike other disciplines in industry, results of decisions can then be predicted and validated.

Wang et al. [82] published the «*Filling-the-Gap*»-tool that, based on resource-level monitoring data estimates performance parameters of software performance models. The performance parametrizes queuing network models that are centred around resources. The tool supports four different algorithms for calculating the parameter estimates, whereas all algorithms depend on response times of queue lengths. The *Filling-the-Gap*-tool is constituted of four main components: (1) *FG Local DB* - a database storing the monitoring data, (2) *FG Analyzer* - calculates the parameter estimates with one of the available algorithms, (3) *FG-Actuator* - that updates the parameters in models, (4) *FG Reporter* - that provides feedback in form of reports for the developer. The *Filling-the-Gap*-tool is based on the design outlined in [61]. In this work, the authors describe the parameters that are estimated in more detail. The parameters the tool is capable of estimating are: (1) the *population*, (2) *resource consumptions*, (3) *think times* and (4) *stage durations, transition probabilities, efficiency.* A *stage* thereby describes the state of resources. The possible stages are: start-up, failure, low- and high contention.

5.8 Other related work

In the performance engineering discipline on a wide range of tools can be set up. Queuing models are popular notations for describing and analysing performance models. In [12] a list of queuing theory software is available.

Woodside et al. [86] outline that SPE is a software engineering discipline. In software engineering, different graphical- or formal models may be used. In this context, it is necassary to distinguish between two types of models: (1) *software models* that have their focus on the functionality of a software system and (2) *performance models* that are based on resources and the resource usage.
Furthermore, Smith et al. [76] describes two types of performance models: (1) *Software execution models* and (2) *system execution models. Software execution models* are based on *scenarios.* A scenario is used to model a workload that a specific interaction behaviour results in. *Software execution models* are described using execution graphs. They are used to evaluate one specific workflow and unveil critical design limitations. Execution graphs therefore show the utilization of platform resources of process steps.
System execution models on the other hand are modelled with some sort of queuing model whereas queuing network models or layered queuing network models are used. *System execution models* may also be simulated. They are used to analyse a software system under load. That means many different *scenarios* are executed simultaneously.

Franks et al. [85] state that there are two approaches in software performance engineering: the predictive model-based and the measurement-based approach. They define software performance engineering as follows: »*Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance*

requirements« [85]. Furthermore, the authors suggest that there is a need for combining the model-based with the measurement-based approach for the performance evaluation of computer systems and networks. They also argue that empirical measures can be integrated in the predictive models over time and re-estimate these models.

Moreno et al. [60] describes three distinct stages of evaluation of performance capabilities in software performance engineering: (1) The *Simple-model-*, (2) the *Best- and Worst-Case-* and the (3) *Adapt-to-Precision Strategy*. They argue, that a simple model is used at first and then over time it is refined and extended. A concrete workflow for this approach is however not available.

Cala et al. [63] outlined the possibilities that OASIS TOSCA brings for reusability and portability of Cloud applications. They argue that the TOSCA specification can fulfil the requirements towards a scientific workflow and furthermore introduce additional potential benefits such as »*portability, automatic deployment and scalability of workflows* « [63]. Breitenbücher et al. [83] showed an approach to transform DevOps artefacts to TOSCA. They distinguished between *Node-centric artefacts* and *Environment-centric artefacts*. The purpose of environment-centric artefacts is the modelling of multi-node environments including the relations between nodes. On the other hand Node-centric artefacts are used to model single nodes. Clearly in practice, both types of artefacts supplement each other. In their work, they proposed a framework to transform artefacts of type Chef Cookbooks [4] and Juju Charms [10] to TOSCA artefacts.

Brambilla et al. [34] argue that model-driven engineering increases the efficiency and effectiveness in software engineering. The authors suggest a significant growth in usage in the future. One of the reasons for that is the convergence of business analysis and software development.

Humble et al. [48] argue that in order to continuously deliver software, the deployment process must be highly automated. Tasks like the deployment of the artefacts to an operation environment became routine tasks and thus a high degree of automation is needed.

The Architecture Tradeoff Analysis Method (ATAM) was introduced by Kazman et al. in [52]. ATAM extends SAM, the Software Architecture Analysis Method [51].
ATAM is a method used very early in the requirements- and design-phase of a software development process. One of the reasons for using ATAM is to build understanding amongst stakeholders of a software project about the design alternatives. Furthermore it aids in evaluating a software systems architecture trade-offs concerning their competing quality attributes such as performance and security. To achieve this, the authors originally proposed a spiral model with six phases. Since then, the ATAM was reworked and now has nine phases. More information about the current ATAM can be found in [2].

CHAPTER 6

An approach for Agile Performance Engineering

6.1 Overview

In the last chapters, the theoretic background and related work was presented. It is now time to introduce my contributions to this research domain, especially the *ASPE*-approach. The underlying assumptions, that the proposed approach is based on as well as hypotheses that are evaluated in Chapter 7 are expressed as follows:

Hypothesis 1. *The agile methodology in combination with continuous delivery, model-driven engineering and the use of the Markov model formalism for describing workloads on resources is a valid approach to combine predictive- and measurement based approaches in SPE.*

Hypothesis 2. *The transition probability between pages/features of a typical Web 2.0 application can be described in Markov model notation and in consequence fulfills the Markov assumption.*

Assumption 1. *Typical Web 2.0 applications are very likely to be expressed in terms of ergodic systems. And if not, they can be transferred to such.*

Hypothesis 3. *Users in typical Web 2.0 applications are not bound to a specific starting state. Users are likely to enter at any state.*

Hypothesis 4. *The fraction of users in each state in the long run, given that the Markov assumption holds, will be close to the steady state solution of the user behaviour expressed in Markov model notation.*

Hypothesis 5. *In order to retrieve service times of operations on resources, it is sufficient to measure response times on a language level.*

For evaluating the proposed hypotheses, a case study was carried out and is outlined in Chapter 7. The case study is based on the concepts, approaches and methodologies outlined in the following sections. The proposed approach as well as the hypothesis are evaluated for a typical Web 2.0 application. However, the ASPE-approach may also be valid for other types of software systems.

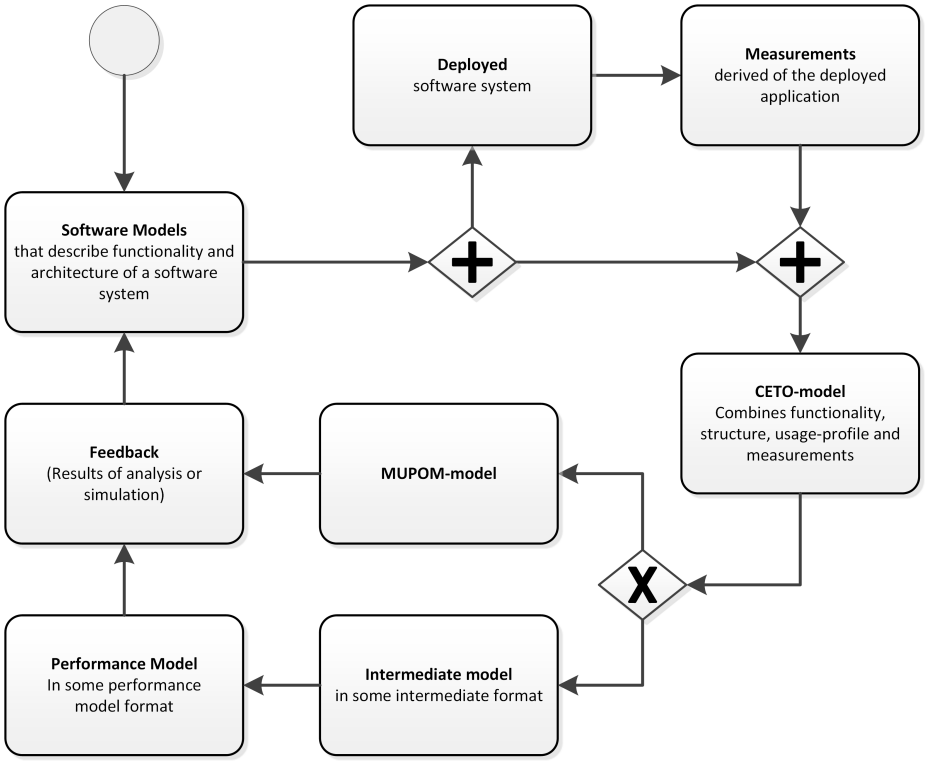

Figure 6.1: The model-measure-feedback cycle is the basic workflow of the ASPE-approach.

This chapter is structured as follows: First, a methodology in the field of Software Performance Engineering that combines the model-based- with the measurement-based approach is proposed. Specifically, a workflow-model and two meta-models for integrating predictive-, static- and measured data are proposed. In consequence, the proposed workflow integrates modelling, measurement and analysis tasks into an agile software performance engineering process, the *ASPE*-process.
The two proposed metamodels are class diagrams in UML-notation, namely the *CETO*- and the *MUPOM* metamodel.
A CETO-model is the central data format in the ASPE-process for capturing static, predictive- and measurement data relevant to the performance of a software system. Three different approaches are outlined for gathering the needed information of CETO-models (Section 6.5.2).

The MUPOM-model is a performance model used to express user-behaviour with the Markov-model formalism. The user-behaviour combined with the workload-intensity is the *workload* of a system and in consequence describes the arrival rates on resources with respect to queuing networks.

The ASPE-process is an approach that explicitly tackles performance engineering during development and is a hybrid between early-cycle predictive- and late-cycle measurement SPE techniques. The basic workflow of the ASPE-process is shown in Figure 6.1, and in Figure 6.4 it is outlined in more detail. The required types of information and model-elements are described in Section 6.2, based on this, Section 6.3 shows the two metamodels CETO and MUPOM. Section 6.4 describes the ASPE-process in more detail. Approaches to capture measurement-information of deployed software systems is given in Section 6.5. Transformations between the respective models shown in Figure 6.1 are described in Section 6.6. Section 6.7 describes a transformation from Markov models towards queuing networks with respective arrival rates to stations.

6.2 Information requirements and basic elements

This section describes which information and what kind of elements are needed in order to *adequately* build a performance model of a software system. It is the basis for the ASPE-process as well as the two proposed metamodels CETO and MUPOM.
What is meant by the term *adequate* here, is derived from research in the fields of performance analysis frameworks and intermediate model formats with respect to queuing theory and will also be subject of evaluation in Chapter 7. In Chapter 5, different formats and formalisms, namely PMIF, CSM, KLAPER, LQN, PCM, UML SPT and UML for MARTE as well as performance engineering approaches with regard to queuing theory and Markov models were outlined. All of these models and methodologies have more or less fine-grained elements for modelling a software system and its characteristics relevant to performance analysis. These models and methodologies all have a slightly different focus. However, all these formats and formalisms have common elements.
Proposition 1 is derived by findings in literature, especially CSM [62] and MARTE [23].

Proposition 1. *The essential and most basic elements for describing the performance characteristics of a software system are (1)* ***resources*** *that offer computing capabilities, the (2)* ***workload*** *that describes how the resources are being used by users and the (3)* ***workload-intensity*** *that describes the intensity of workloads in terms of inter-arrival times.*

The performance of a software system is in essence determined by resources and the way they are used. Waiting lines appear if the arrival rate is higher than the service rate in a specific time interval. In terms of software systems: A queue on a resource appears if there are more operations than computing power in a given time interval. If an operation waits on another operation to finish this leads to virtual queues on a higher level. Therefore, besides arrival-rate and service-rate on resources, dependencies between operations and

in consequence resources, also determine the performance quality of a system. The *workload* is constituted of the usage-profile which describes the user-behaviour in a probabilistic way including *think-times*. In the ASPE-approach, user-behaviour (and thus the workload) is described using Markov models. The relevant elements to describe a workload with Markov models is outlined in the MUPOM model. *Think-times* are the amount of time that users are in a specific state of the Markov model before switching to another state described by a probability distribution (a normal distribution here). The *workload-intensity* may vary over time and may be time dependent. An example for this is increased traffic at web-applications in the evenings.

The term *user* is used in this work for both, humans as well as other systems that interact with a software system. If a software system does not have user-authentication, in general it still is possible to distinguish between different users by using other techniques like IP-Address/Session/Thread/Cookies/ ...

In this work, the model-based and the measurement-based approach in SPE are combined. Therefore, three types of information are needed: (1) static information, (2) predictive information and (3) measurement information.
In order to combine the model-based and the measurement-based approach in SPE and with respect to Proposition 1, there are four needed information-aspects described in Proposition 2:

Proposition 2. *For describing resources, workloads and workload-intensities, and in order to combine the model-based and the measurement-based approach in SPE, static-, predictive- and measurement information is needed. The following informations are sufficient: (1) Functionality of a software system and the causal ordering of operations that enable this functionality, (2) Structure and topology of a software systems components and resources (3) Predictive- and measurement data of behaviour and in consequence workloads, (4) Predictive- and measurement-data of operation durations on resources.*

Aspects (1) and (2) are static informations about a software system. Aspects (3) and (4) are a mixture of predictive- and measurement data. As measurements are retrieved at the end of each development sprint in the ASPE approach, the higher the degree of completion of a software system in terms of development, the more measurement-data is available and the less predictive data is needed. Additionally, static performance goals and constraints may be defined on a use-case- and an operational level. Besides feedback, also checks whether these performance constraints or goals are violated or are predicted to be violated can be made at the end of every sprint.
There are two information requirements for measurements: user-behaviour and the duration of operations on resources. Measurements can only be retrieved from observable emissions, hereafter called *observations*.

The modelling of resources and their workloads as well as the causal ordering of operations leads to queuing networks with nested requests and thus simultaneous resource possession. The outlined information-aspects are capable of taking extended properties of resource

usage into account. As described earlier, it is rare in software systems to only utilize one resource per request without operations depending on each other. There is a myriad of applications in the SPE-research available that make use of queuing networks. Most of these approaches are based on the product-form condition of queuing networks [25]. The LQN-formalism [43], as an example, was specifically designed for describing and analysing layered architectures with nested operations and thus simultaneous resource possession build upon the theory of extended queuing network [70].

The way a software system is used, is described by a *workload*. There are usually many different ways how a software system is used in practice. Therefore, the workload may be substituted by multiple *usage profiles*. A single *usage profile* here is expressed by sequences of use-case activations. The time between two use-case activations is expressed with *think times*. Think times are described with probability distributions. *Workload-intensities* are on top of a usage profile and describe the amount of users that interact with a system in a specific time interval. Also *workload-intensities* are described using a probability distribution.
In order to deliver the functionality described in a use-case, a software system has *operations*. It must be expressed what operations and what resource usages a usage profile results in.
Another important aspect is to describe *how long* an operation on a resource takes. For example, when a user logs into a system, first this user has to provide its username and password. The software system then has to retrieve this user from a datastore, for example a relational database. It is essential to know how long this operation takes.

Proposition 3 summarizes the described findings above. It is essential to point out that it has to be verified for every software system of analysis if the Markov-property holds for describing the user-behaviour. Furthermore it has to be verified if the product-form conditions for a specific system holds when analysing a systems performance with product-form queuing network solution approaches. Based on that, either closed-form solutions or approximative approaches may be used.

Proposition 3. *Following elements are sufficient for analysing the performance of a software system given a user-behaviour where the Markov property holds: (1) Use-cases and their needed operations (2) physical resources, (3) Causal ordering of operations on resources, (4) a Markov-model describing user-behaviour, (5) workloads of user-behaviour, (6) think times of users between use-case transitions, (7) operation durations.*

6.3 The CETO- and MUPOM metamodels

In order to represent all necessary information and basic elements for evaluating the performance of a software system with the ASPE-process, the following two metamodels are described here:

1. CETO - ***C****omponents* ***E****mission and* ***T****imely* ***O****bservations.*
   This model captures static-, predictive- and measured informations about a software system.

2. MUPOM - ***M****arkov* ***U****sage* ***P****rocess and* ***O****peration* ***M****easurements.*
   This metamodel describes user-behaviour using the Markov-model formalism as well as operations on resources. *MUPOM* is a performance model.

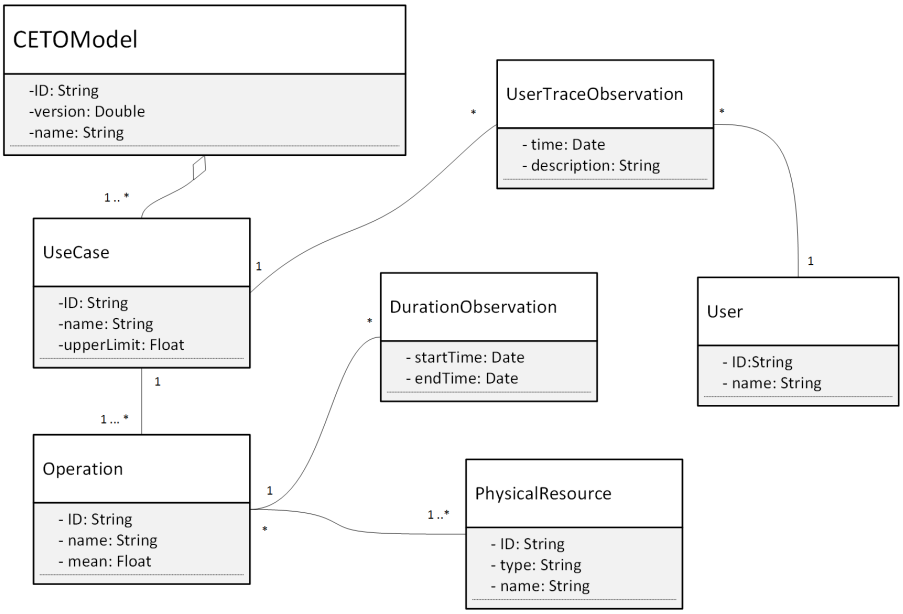

Figure 6.2: The CETO metamodel.

CETO is the central model in the ASPE-process used for capturing all relevant information outlined in the previous Section.
The reason for proposing two metamodels is two-folded. The ASPE-process combines the model-based- and the measurement-based approach in SPE. To achieve this, a mix of static-, predictive- and measurement data is needed. Models such as CSM or PMIF provide capabilities to model static- and predictive data, but no measurements. Therefore, the first and foremost reason for the conception of the CETO-metamodel is to also capture measurement-data. The MUPOM-metamodel is designed with the goal

to model stochastic processes of user-behaviour in Markov-model notation. There is, to the best of my knowledge, no such metamodel in the SPE-domain available. The CETO-model as well as the MUPOM-model are *living artefacts* within the ASPE-process. The CETO-model is *self-contained* which means that it contains sufficient information to analyse the performance of a software system. The CETO-metamodel can be found in Figure 6.2 and is divided into two parts:

1. Static-information: Functionality, structure and topology of a component-based software system.
2. Predictive- and measured information of user-behaviour and operation durations.

The MUPOM-metamodel is shown in Figure 6.3. The contained elements can be classified as follows:

1. Resources, use-cases and operations on resources.
2. User behaviour denoted in Markov-model notation.

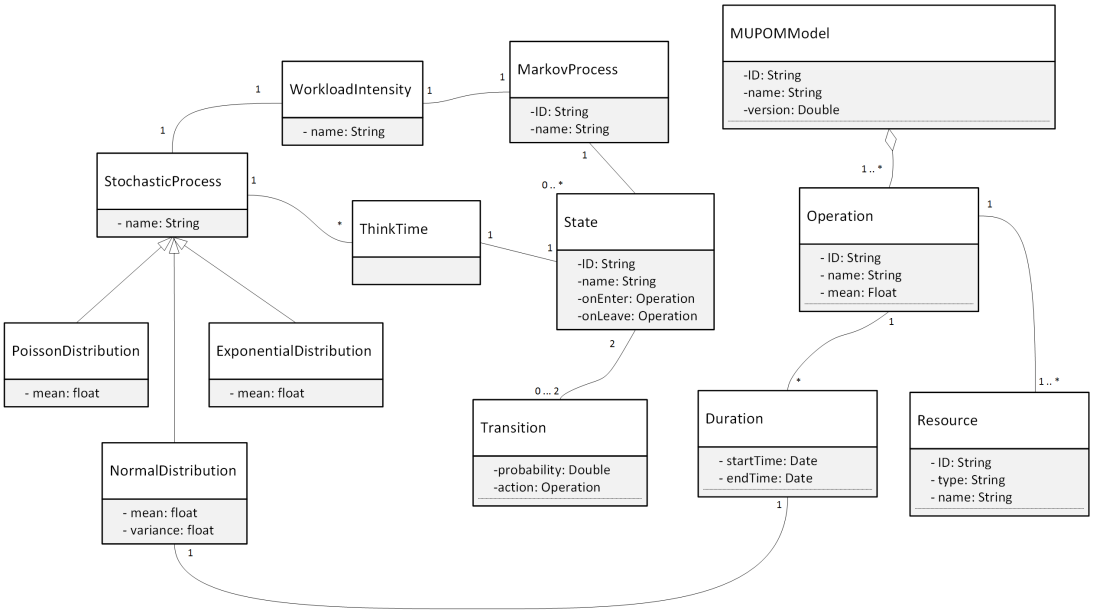

Figure 6.3: The MUPOM metamodel.

The MUPOM-metamodel defines only *open workloads* with external arrivals and departures. Internal arrivals result from use-case dependencies on its, possibly nested, operations. Therefore the arrival-rate of resources is determined by external- and internal arrivals. The arrivals are modelled using one of three available distributions: Poisson,

Exponential- and Normal distribution. *Duration* and *Think-time* also follow one of these three distributions.

A MUPOM model can be seen as the result of *solving* a CETO model. A methodology for doing this is described in Section 6.6. The MUPOM metamodel does not take nestings, joins or other enhanced operation-sequences into account. This is due to the goal of the ASPE approach of being usable in practice with low overhead and high benefits. However, this is a possible future extension for further research. In both metamodels, one operation always utilizes one or more resources.

6.4 The ASPE-process

The ***A****gile* ***S****oftware* ***P****erformance* ***E****ngineering*-process (ASPE) is built upon the agile principles of software development. Therefore, software development is concerned as being created in an incremental and iterative manner. Parts of a software system are continuously integrated and in consequence continuously delivered to a test- or production environment where the running software is profiled. The ASPE-process covers the software life-cycle phases *Requirements, Specification, Design, Implementation, Integration* and *Testing* as defined in Section 2.2.
Design- as well as implementation decisions during development may have a strong impact on the performance of a software system. The ASPE-process ensures fast feedback for developers and designers and forces checks if QoS- or SLA- constraints are violated or are predicted to be violated.

The software-, CETO- and MUPOM models are *living artefacts* that are adapted over time. Model-driven engineering is used throughout, respective model-transformations in the ASPE-process are: (1) *Use-case and operation topology to CETO*, (2) *CETO to MUPOM* and (3) *MUPOM to performance model*. These transformations are described in Section 6.6.

As outlined in the introduction of this thesis, Software Performance Engineering is not necessarily part of software development in practice. Instead, after development when performance problems occur, these problems are mitigated in an ad-hoc trial and error fashion. I assume that a reason why SPE often is neglected in practice is the tremendous overhead that it brings. In consequence, the ASPE-process tackles this in three ways: (1) Force automation, (2) Reuse artefacts and (3) Use approximations instead of over-engineering SPE. The ASPE-process was in consequence also designed with respect to the agile-principle: »*working code over extensive documentation*« [16]. Automatically, as much information is gathered as possible from observations, instead of the extensive need of full-grown, predictive diagrams. The workflow of the ASPE-process is shown in Figure 6.4.

The information requirements to adequately build a performance model are outlined in Section 6.2. To represent the static information concerning functionality, structure and topology of a software systems operation two types of models are sufficient: (1) A

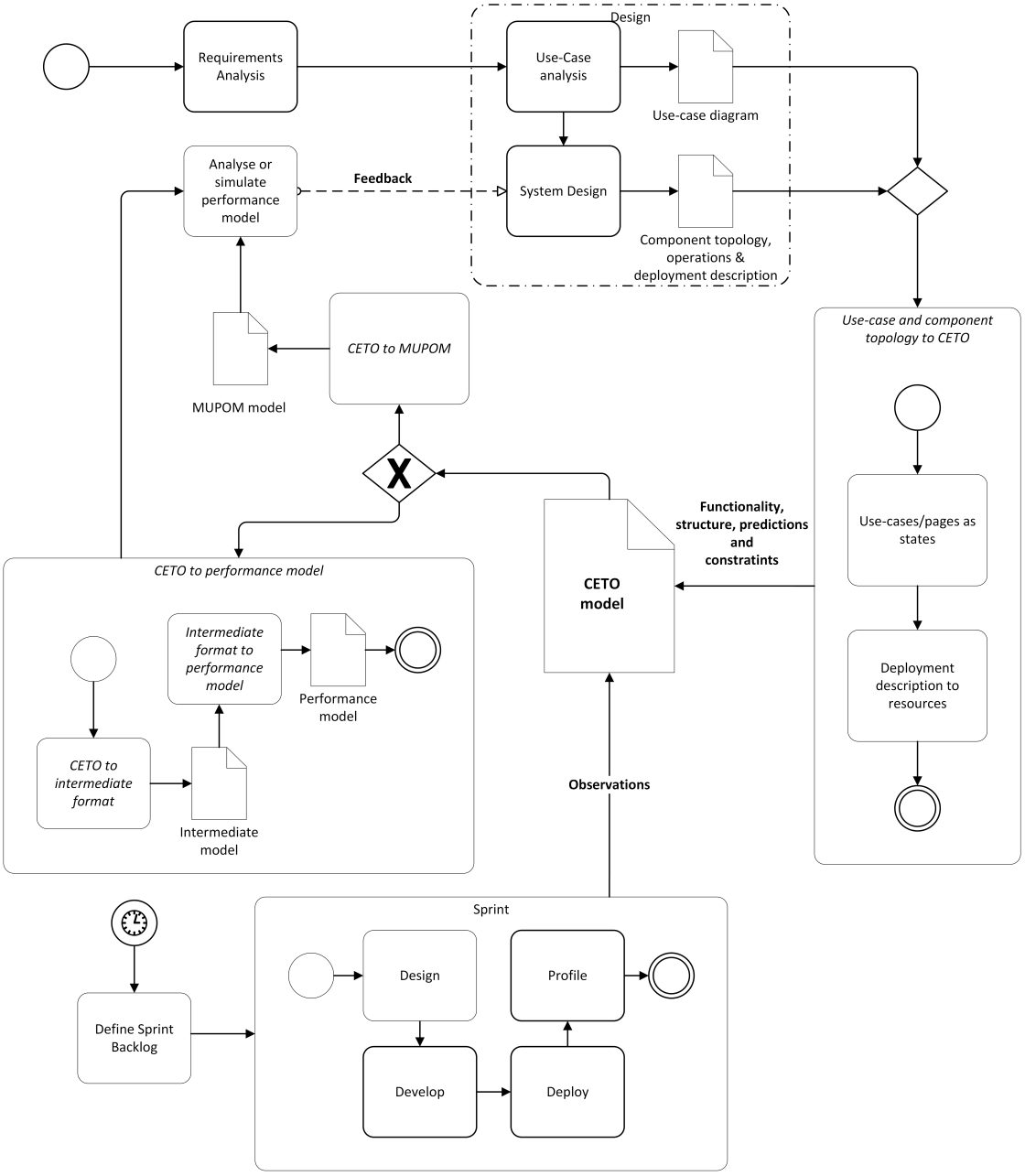

Figure 6.4: The ASPE-workflow that combines the model-based approach and the measurement-based approach.

Use-case diagram and (2) a component topology model that describes the structure of components as well as their deployment and therefore also the dependencies amongst components and resources. Based on these models, transformations may construct the basic structure of Markov models. One state in the Markov model represents one use-case.

A CETO-model is the central element in the ASPE-process. All necessary information is kept in a single CETO-model and *solved* by transforming it to a MUPOM-model. The CETO-model is over time enriched with observations. The ASPE-process defines a feedback loop, therefore CETO-, MUPOM- as well as the software models are *living artefacts* that are changed and adapted over time.

After every development-iteration (*Sprint*), the software system is deployed and profiled. There are two types of observations that enrich the CETO-model:

1. The tracked observations of use-case and operation calls.
   A use-case has operations that *use* components and in consequence resources. Use-cases, operations as well as components and resources generate emissions when being *used*. These emissions are observed through log-files, profiling tools or similar.

2. The duration times of *operations*.
   In order to provide functionality, a use-case performs *operations*. These operations result in computing times on resources and are measurable.

The ASPE-process combines the model-based and the measurement-based approach which was the main goal of this theses.

6.5 Profiling

In order to integrate measurements into performance analysis and enhance the accuracy of prediction over time, observations are captured on the deployed software system. There are two types of informations that are observed in the ASPE-process and stored in a CETO-model: (1) Operation durations on resources and (2) workloads by observing user-behaviour.
For capturing these measurements there are different tools and techniques available. All of these approaches demand different types of information. They mainly differ on the level the information that is extracted.

6.5.1 Resource operation duration measurements

For any operation that demands a resources service, a time interval can be defined in which a resource is *possessed*. From a hardware point-of-view, an operation is not necessarily carried out at once. Instead, slices of computing time will be used [43]. However, a resource operation duration is described by a number representing the total time it took to process an operation at a resource. The time interval for an operation may vary from one user to the other or even for one user when the user carries out an operation multiple times. An example for this is the upload of a picture in the Travelistr-example. Obviously, the time of uploading and processing an image depends on the size of the image and therefore varies. Having many operation durations observed allows to build a probability distribution on the duration.

In [59], the duration of operations is described by *service demands of requests*. A system has different *classes of requests* and is therefore called a *multiclass QN model*.
The duration of operations can either be directly observed gathering complete information by respective profiling tools such as MagPie [27], Kieker [68] or simply by logging. Other techniques include statistical inference techniques such as linear regression or a maximum-likelihood method outlined where only incomplete information is available [53].

6.5.2 Workload and workload-intensity

The workload is derived from profiling user-behaviour and is constituted of the usage-profile, including think-times on use-cases and the transition-probabilities between use-cases. The workload-intensity is how often a specific behaviour is performed in a time interval. The workload-intensity may vary over time.
For modelling user-behaviour, the mathematical formalisms of Markov models are used in the ASPE-approach. Of special interest here, as will be outlined later, are furthermore Hidden Markov Models (HMMs). The use of HMMs in SPE is relatively new but may have increased usage in the future [41].

Remarks:

1. Some single observations of users are probably too little to describe the overall behaviour of users. The sample should therefore be *big enough*.

2. Modelling user-behaviour with Markov models only brings reasonable results if the Markov-property on user-behaviour holds.

There are several approaches how the user-behaviour can be captured. Please mind that tools are available that use only information on one specific layer or combined information from different layers. The CETO-meta-model is able to capture information with all the three approaches outlined as follows.

Approach 1 - Request-based capturing

This approach is very simple and straightforward. The usage-profile is derived from request calls. A request is the triggering event of a use-case and therefore one state in the Markov model.
The needed measurement data for this approach are request calls plus the time the request entered the system and the time a response left the system. The user that made the request and operation duration measurements for every request call can be calculated using statistical inference or similar techniques (e.g. [53]). The time between a system sends the response and the user send the next request is the *think time* in a specific use-case and therefore in a state of the Markov-model.

Approach 2 - Component-based capturing

Here the requests are not directly observed, instead the emissions of components are observed. A component is a logical resource and every logical resource utilizes a physical resource at some point.
The state sequence and the transition between states cannot be observed directly, the underlying Markov model therefore is *hidden*. Only the emissions of the components are visible. The underlying Markov model can be unveiled using the mathematical formalisms of hidden Markov models.
In order to be feasible, there must be a correlation between operation emission and the respective use-cases. The correlation must be significant in a sense, that an assignment towards the use-cases is possible with a specified statistical significance threshold. This may not be the case for every software system.

In this approach, the information is again profiled per user. The profiling result will look somehow like the following example, whereas James, Linda and Patricia are users and AC,BC,DC, ... are emissions of components grouped per request.
$O = \{\{(James) : \{IC, DC, AC, TC\}, \{EC, IC, PC, RC\}, \{DC\}, ...\},$
$\{(Linda) : \{BC, RC, IC\}, \{LC, AC\}, ...\},$
$\{(Patricia) : \{IC, AC\}, \{OC\}, \{LC\}\}, ...\}$

The notation for HMM used in this work is outlined in Section 3.4.2.

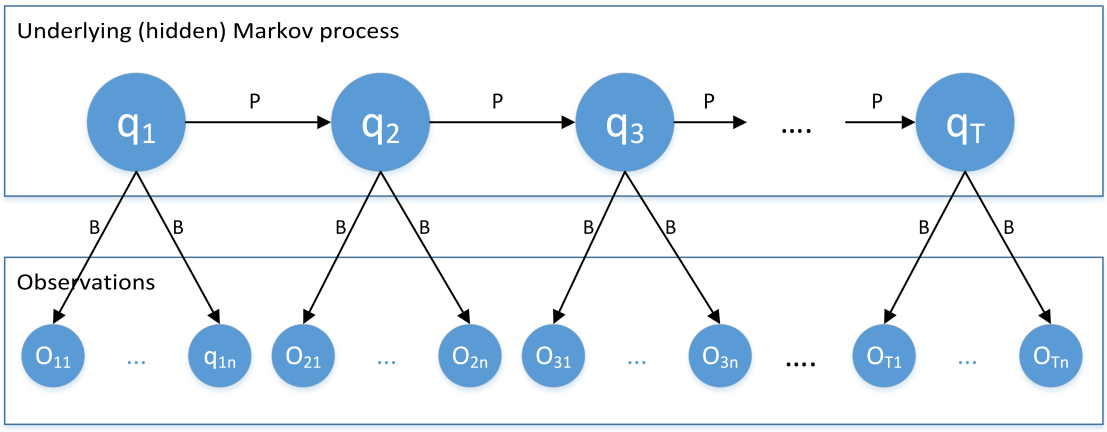

Figure 6.5: Hidden Markov Models emissions and states.

This leads to the definition of the main elements in the HMM and their correspondence to use-case- and component topology descriptions. As stated earlier, there must be a probabilistic relationship between emissions of components and states of the underlying Markov chain. This relationship is encoded using the *observation matrix* $\mathcal{B}$.
In Figure 6.5 the relationship between observed emissions and the underlying Markov-process is depicted.

HMM element	Software system element
T	The observation sequence length is the total number of resource operations where emissions were tracked.
N	The total number of states in the underlying Markov chain is the total number of use-cases.
Q	The set containing all states is constituted of all use-cases.
q_t	A single state is represented by a single use-case.
M	The number of distinct observation symbols is the number of distinct resource operation emissions.
$\mathcal{O}$	The observation sequence is represented by all observer resource operation emissions.
O_j	A single observed emission is represented by a single resource operation emission.
$\boldsymbol{P}$	The transition matrix is constituted by the transition probabilities from one use-case to another.
$\mathcal{B}$	The observation probability matrix is constituted by the probabilistic distributions that link resource operation emission and use-case.
π	The initial distribution of the underlying Markov chain is the initial distribution of believed use-case usage.

Table 6.1: Corresponding HMM and software system elements for unveiling Markov models given incomplete information.

In case of software systems it is very unlikely to only use exactly one resource per use-case. Imagine an Image-Upload in the Travelistr-system: Simultaneously the Database, the Application-Server and the Image-Server are used. This concept of simultaneous resource possession was already examined and described in the LQN-metamodel by Franks et al. [43] (see Section 5.2.4 for more details). Of special interest here is the fact, that the possession and usage of multiple resources also leads to emissions of these resources. Therefore the hidden Markov model here has to relate a set of emissions to the underlying Markov process, using the observation matrix B. This is a major difference to classic hidden Markov models. Therefore special techniques for unveiling the underlying Markov process based on this emissions has to be taken (see Section 6.6 for more details.) In conclusion, Table 6.1 summarizes the respective equivalent elements.

Approach 3 - Resource-based capturing

This approach does profile the emissions on a resource level. On the resource-level it is to my knowledge not possible to get the user information that requested a specific service and therefore it can not be distinguished between different users or any other Session information. However, also here the use of HMMs seems an interesting approach and a very interesting domain for future research.

6.6 Model transformations

The ASPE-process is model-driven and therefore model-to-model transformations described in this Section. First, the transformation towards a CETO-model is described including the approaches to capture measurement-data. Second, a transformation methodology from CETO to MUPOM is outlined. Third, the alternative way using CSM as intermediate format and from here transforming to a performance model is outlined.

6.6.1 Software models and observations to CETO

This Section describes how information from two different types of sources, namely software models and profiling-data, are combined and integrated in a single CETO-model. The different sources provide static-, predictive- and measurement information.
As described in Proposition 1, the needed information is divided into resources, workloads and workload-intensities. Proposition 2 consequently outlines necessary elements with regard to the ASPE-process that take predictive model-based elements as well as measurement data into account. The CETO-metamodel expresses these aspects in its conception. This Section will be described four layered, describing the four types of relevant information for evaluating a software systems performance in Proposition 2.

1. **Functionality and operations**
   Use-cases and respectively features describe the functionality of a software system. They may for example be derived from use-case diagrams. Use-cases define available ways for using a software system from outside through defined interfaces. In order to provide functionality, a software system has components which in turn provide operations.

2. **Structure and topology of components and resources**
   Software components interact with other software components or use resources. Information about components, their structure and dependencies on resources or other components is derived from a component-topology diagram.

3. **Resource operation durations**
   At early stages of a software project or the start of development, only predictive information about operation durations are available. As time goes by, the predictive information is replaced by measurement data gathered as described in Section 6.5.

4. **Workload and workload-intensity**
   Similar as the resource operation duration, at an early stage of a software project, there is no profiling data available. Therefore, predictive data has to substitute the not yet available measurements. As time goes by, the predictive information is replaced by measurement data as described in Section 6.5.

6.6.2 CETO to MUPOM

This Section describes the transformation from CETO to MUPOM. CETO models contain static- and predictive information as well as measurements. The observations in the CETO-model have a structure shown in Figure 6.6.

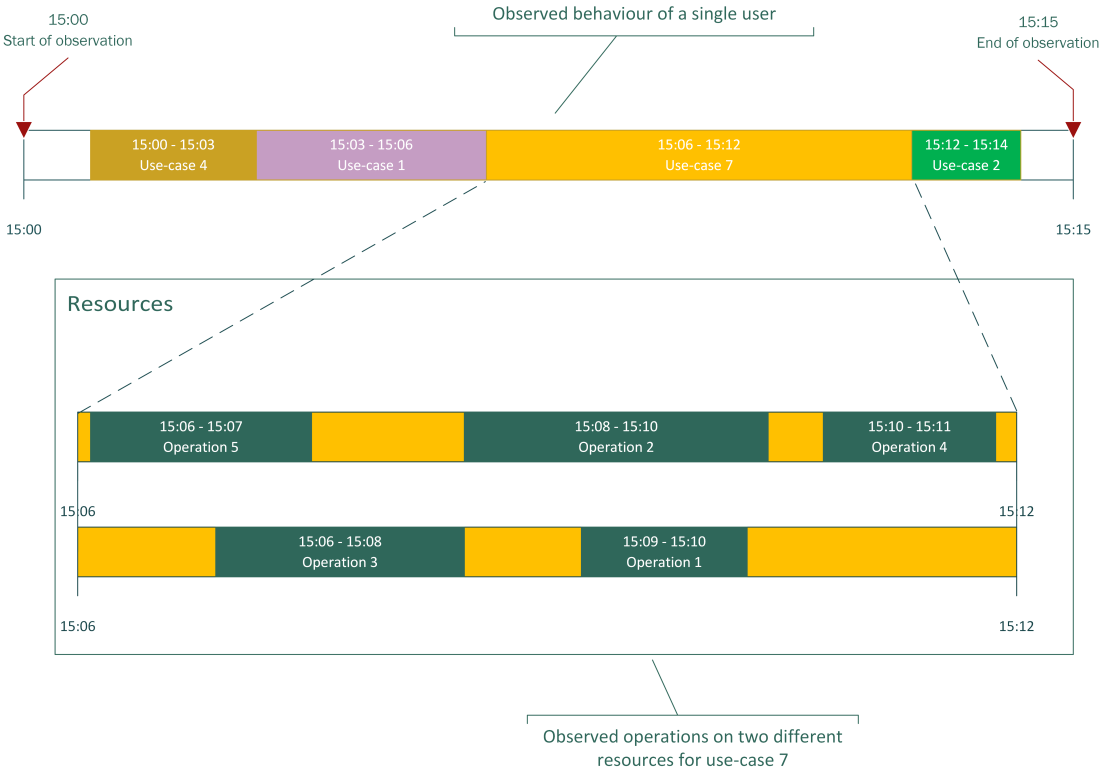

Figure 6.6: Use-case requests and operations on resources resulting in two layered observations.

For transforming a CETO-model to a MUPOM-model, multiple steps are necessary. As stated earlier, a MUPOM-model can be seen as the *solution* of a CETO-model. The steps to *solve* a CETO model are as follows:

1. **(Optional) Classify users and group users of same types.**
   The classification of users must be defined specific to a software system. It does not make sense in a general classification. For example in the Travelistr-example there may be two different groups of users: People that just browse and look at photos from others and people that upload a picture.

2. **(Optional) Unveiling the underlying Markov chain**
   If the observations are captured according to approach 2 or 3, first the hidden Markov model must be unveiled based on the emissions.

Rabiner et al [66] outlines three basic problems of HMMs that must be tackled in order to be useful in practice. For unveiling the underlying Markov model, Problem three is of interest here:
Construct a model λ given the observations $\mathcal{O}_T$ where $\Pr(\mathcal{O}_T|\lambda)$ is maximized.
In Section 3.4.2 the theoretic background of HMMs was briefly outlined. It was outlined, that HMMs are full probabilistic models. To solve problem three, dynamic programming is useful and in general the Viterbi-algorithm can be used. However, as shown in Figure 6.5 as a Use-case usually results in more than one emission, special algorithms need to be used. Li et al. [54] define an approach for solving problem three when having multiple emissions per state.

3. **Canonicalization of operation-durations**
   Given multiple operations on a resource, in fact, these operations are computed by the resource in a time-sharing fashion. Instead of computing an operation at once, time-slices are assigned to an operation. For the performance evaluation of a system, this technique is not of interested and therefore is not contained in the performance models. Instead, the total amount of time to compute an operation is modelled. Therefore, operation-duration observations must be transformed into a canonical form.

4. **Construct probability distributions for think times and operation durations.**
   Think times as well as operation durations are described by a normal-distribution. A normal-distribution has two parameters: the mean λ and the variance σ^2.

5. **Constructing the transition matrix P of the Markov model.**
   A Markov model based on the observations (User-Traces) is constructed by calculating a transition-matrix in the discrete case and a transition-rate matrix in the continuous case.

6. **Check whether the Markov-property for the usage holds**
   This check may be done similar as described with the approach in [55]. The Markov property is validated approximately by comparing the history-independent transition probabilities with the history-dependent transition probabilities. If the history-independent does *not much* diverge from the history dependent, the Markov property is assumed to hold. Otherwise the usage profile can not be modelled accurately with Markov models. In that case, a transformation from CETO to some intermediate format as described in Section 6.6.3 can be considered as an alternative.

7. **Calculate the steady state distribution**
   If a unique limiting distribution exists, this is the steady state that describes the long-run behaviour of a system independent of the starting state. For calculating the steady state, approximative solutions ma be considered. This step may also be subject of future research as outlined in the next Chapter.

8. **(Optional) Derive the causal orderings and parallelism of operations.** In order to be practically usable by reducing the overhead of Performance engineering, the ASPE-approach and thus the CETO-metamodel do not consider complex sequences like splits, forks, .. for operations on resources. However, as outlined in the next Chapter, this is also another interesting research direction to retrieve operation sequences and orderings from observations.

9. **Check whether the system satisfies the product-form assumption of queuing networks.** If it does satisfy the product-form assumption, a wide range of possible closed-form solutions are available. If it does not, approximative approaches like LQN for example might be more suitable.

10. **Construct a performance model.** In Section 6.7 an approach to derive queuing networks from Markov models that describe the user-behaviour is outlined. However, if the product-form assumptions or other prerequisites like ergodicity do not hold, other formalisms outlined in the previous Chapter may be considered.

6.6.3 MUPOM to some performance model

As shown in the previous Chapter, there is extensive research in SPE available. Futhermore, many different *intermediate formats* for performance engineering exist. The respective models showed applicability in practice. For the transformation towards a performance model, two steps are necessary: (1) An intermediate step towards an intermediate format and afterwards (2) a transformation form the intermediate format to a performance model. The descriptions of these transformations are omitted here.

6.7 From Markov models to queuing networks

As stated many times, to capture the user-behaviour in this approach, the Markov model notation is used. Resource utilization in software systems is mainly a result of user-behaviour. The MUPOM-model, besides the Markov model itself, also contains the solution for the steady state if it exists. As described in Section 3.4.1 a Markov chain has a limiting distribution if it is irreducible and all its states are positive recurrent [72] (page 34, theorem 54). For the purpose of transferring a Markov chain to queuing networks with the following approach, the stronger *ergodic*-assumption that demands an irreducible and positive recurrent chain and additional non-periodicity must hold. The reason herefore is that Little's Law (as described in Section 3.5.1) is necessary to determine the number of users in a system. Following Little's Law for ergodic systems, the average number of users in an ergodic system is equal to the arrival rate of users times the average time spent in the system (Equation 6.1).
Given these findings, it is now possible to translate from Markov models and in consequence from MUPOM-models to queuing theory.
With Little's law, we can calculate the average number of users in a system, and with

the steady state distribution we know what fraction of users are in each state (Equation 6.2). Given this knowledge, there are two main possible aspects that we can focus on:

- State-based evaluations
- Transition-rate-based evaluations

As i am not dealing with usability in this thesis and the states of the Markov chain represent use-cases it is not surprising that i am more interested in transition-rate-based evaluations. Transitions between use-cases are triggering-events for resource utilization in software systems.
Markov chains in the MUPOM model are described in discrete time. The time-step is one second. We know how many users are in each state on average. And blessedly, Markov chains fulfil the Markov-property and are therefore memoryless, the transition rate between two states can simply be calculated by multiplying the average number of users with the transition probability (Equation 6.3). The total number of average new entering to a state per second then is the sum of all arrival rates to that state (Equation 6.4). We then know the arrival rates of each state and by taking the inverse of it, we get the inter arrival rate that is very important and together with the service rate the basis for queuing theory analysis.
Following this workflow described verbally is now outlined more formal:

1. Calculate average number of users in system using Little's law:

$$E(N) = \lambda * E(T) \tag{6.1}$$

2. Calculte average number of users in each state

$$N_i = \pi_i * E(N) \tag{6.2}$$

3. Calculate transition rates between each state

$$\lambda_j = p_{ij} * N_i \tag{6.3}$$

4. Sum all incoming transition of each state

$$\Lambda_J = \sum_{j=1}^{k} \lambda_j \tag{6.4}$$

5. Take the inverse of the arrival-rates in order to get the inter-arrival rates

From here on, queuing systems/networks can be constructed in order to evaluate a systems performance. There is a lot of research available herfore. An example is the Mean Value Analysis (MVA) which will be used in the Case-study in Chapter 7. Basic idea there will be the detection and removal of bottlenecks.

CHAPTER 7

Evaluation

7.1 Overview

In the previous chapter, the ASPE-process together with its information requirements and resulting metamodels as well as transformation approaches was outlined.
The main Hypothesis is that the ASPE-process is capable of combining the model-based approach with the measurement-based approach in SPE. The aim now is to evaluate this and all other Hypotheses/Propositions in a case study. The case study covers the implementation of the Travelistr software system using the ASPE-approach.

In the case study presented here, the guidelines of Runeson and Höst [69] are applied.
Using the analytical research paradigm to verify the Hypotheses in an isolated way would not be appropriate here. Instead, it is essential to use an empiric approach that, on the one hand, includes an empiric evaluation of usage-behaviour modelling with Markov models, and on the other hand, an empiric assessment of the benefits, needed tools and the overhead associated to using the ASPE-approach for Software Performance Engineering. This will lead to a better understanding of the investigated phenomenas [69].
The case study is of explanatory as well as of exploratory nature. It evaluates the hypotheses made and furthermore investigates additional phenomena and directions for future research.
The case study both uses quantitative and qualitative data. In order to be as transparent as possible, all results of this case-study are, besides being presented here, also available online together with the implementation of the Travelistr-software system [8].

The case study is carried out in a real-world setting for the Travelistr software project. The project is structured in an agile way with time-boxed iterations that are called *Sprints* here. This naming is due to the use of the agile-methodology SCRUM, a specific agile process model. This iterative approach allows much flexibility in doing this case study. In total, four iterations were carried out, whereas *Sprint 0* exclusively uses predictive

knowledge and is furthermore used to set up the development environment. After each iteration, the current state of Travelistr is deployed to a Cloud platform and predictive- and measurement data in a quantitative or qualitative way are captured and analysed.

This Chapter is structured as follows:
Section 7.2 outlines the objectives of the case study. Section 7.3 defines the way quantitative- and qualitative data is gathered as well as the tools that automate aspects of the monitoring, modelling and load-test approaches within the ASPE-process.
Section 7.4 gives a very brief overview about the Travelistr software system and furthermore outlines the used Technology Stack as well as the Cloud environment it is hosted at.
More detailed information and the results of the iterations are given in the respective Sections 7.5 (Sprint 0), 7.6 (Sprint 1), 7.7 (Sprint 2) and 7.8 (Sprint 3).

7.2 Objectives of the case study

The nature of this case study is both *exploratory* and *explanatory*.

The explanatory part tries to confirm the Hypotheses and goals of this thesis. The main research interest herefore lies in the combination of the model-based- and the measurement-based approach. The ASPE-process is one possible solution to combine them and the results of its evaluation are presented here.
The exploratory part on the other hand found additional hypotheses and aspects as well as future research directions in SPE that are presented within the respective Sections and are summarized in Chapter 8.

7.3 Data collection and Tools

Different data sources were used in this case study. Qualitative data was incorporated from my own experience applying the ASPE-approach in the Travelistr software project. The benefits, tools and the resulting overhead is also being analysed with this qualitative data.
Quantitative data is gathered with respective tools. These tools are:

1. JMVA (a JMT-component) [30]
2. Markov4JMeter [79]
3. *OperationsAndTraceMonitor*
4. *UserTrace2Markov*

JMVA is a software package of JMT and a tool to calculate the MVA-algorithm (see Section 5.6 for more details).
Markov4JMeter is an extension of JMeter [17] used to automatically generate a workload following behaviour described with Markov models (see section 5.6 for more details).

The two self written tools to track and analyse user-traces and service-times are both written in Java. The *OperationsAndTraceMonitor* can be integrated anywhere in the Travelistr-software or any other Java software and writes observations to a CSV-file in a thread-safe manner that has the right format to be evaluated with R-software [64], a statistical computing tool, as well as to the *UserTrace2Markov*-tool, a semi-automatic tool to transfer user-traces tracked with the *OperationsAndTraceMonitor* in CSV-format to a Markov model of Orders one and two.

The analysis of the collected data was done at the end of each iteration.
Additionally, or evaluating the Markov assumption for the usage-behaviour of Travelistr, an empiric test was carried out.

7.4 Travelistr

Travelistr is a Web application that lets travellers share their pictures with others. A detailed description what Travelistr is and what features it offers to its users is given in Section 1.5. The reason for defining a software project such as Travelistr here is, that the Travelistr software system is a relevant and very typical type of system nowadays and stands as an example of many similar systems in the era of Web 2.0.

Technology Stack
Travelistr is a Java EE application realised with Spring-MVC. For database access, the Hibernate framework is used together with the c3pO connection pool. The web-frontend is created with static html-pages and dynamic JSP-pages. Log4j is used as logging framework and *OperationsAndTraceMonitor*-instructions are added at language-level. The Travelistr-App is hosted at an Apache Tomcat 8 server. The data is stored in a Postgresql database and the images are stored using the external vendor Cloudinary [5] which is free up to 75000 pictures or 2GB of storage.

Cloud environment
The Travelistr software system is hosted at the Cloud vendor Digitalocean [6]. Digitalocean makes it easy to create virtual servers, called Droplets, in any of their datacenters around the world. Within a few seconds, the available resources can on-demand be increased- or decreased. For the initial setup of Travelistr, the database is hosted at the London datacenter and the application server runs in a datacenter in Frankfurt. Both Servers are virtual and use the Ubuntu 16.04.1 operating system.

7.5 Sprint 0

In the domain of agile software development, a *Sprint 0* is commonly referred to a time-boxed period that is used for setup- and design tasks. This meaning is also used here. Besides setting up the development environment and the code-bases, also the model-based evaluation and prediction of user-behaviour- as well as system behaviour

is carried out. The result of this predictive evaluation gives a good overview of the characteristics of the Travelistr software system.
In Figure 7.1 the predicted user-behaviour-model is shown and in Table 7.1 this model is expressed with its transition-matrix.

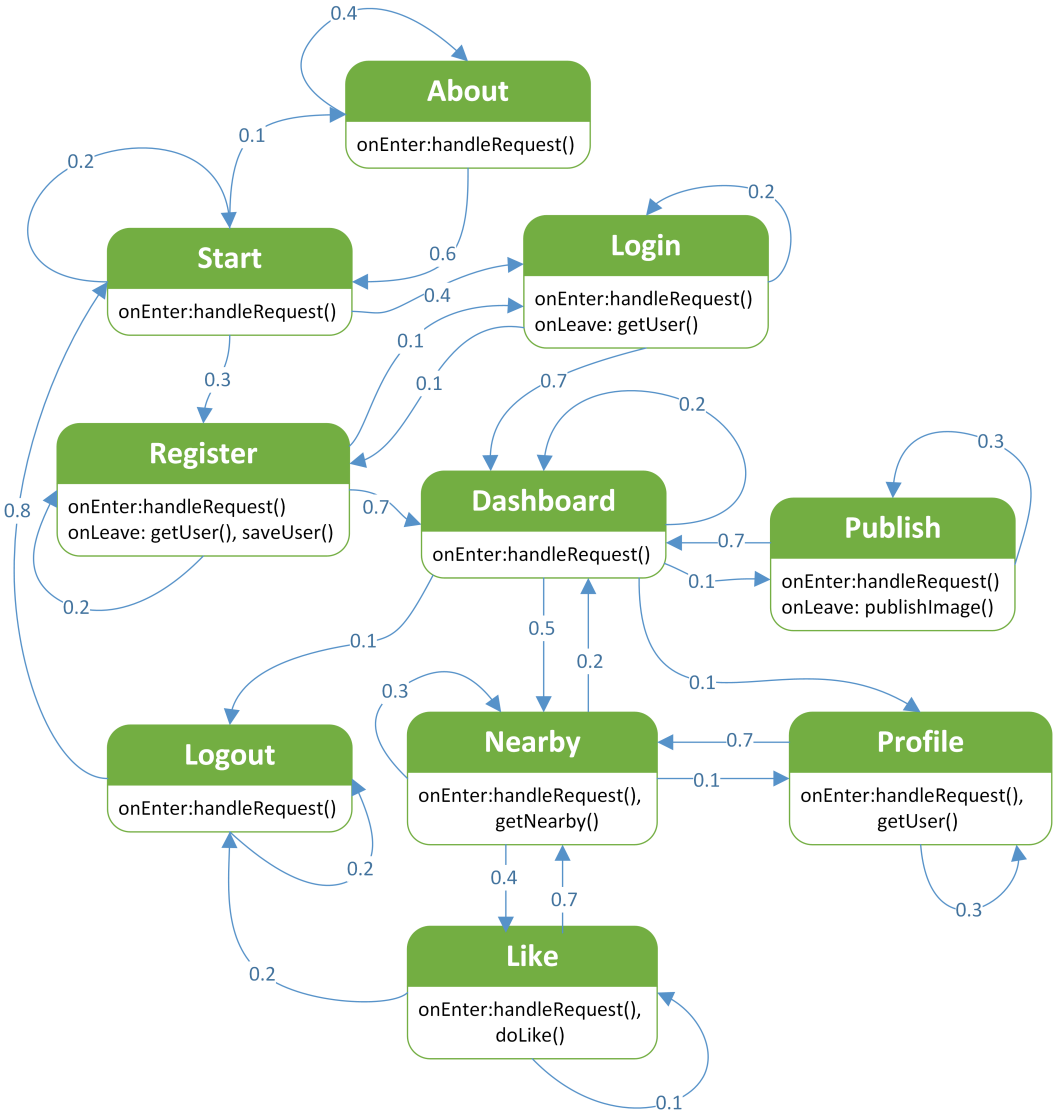

Figure 7.1: Markov model representing the states of Travelistr and the expected user-behaviour defined with predictive knowledge in Sprint 0.

P_{ij}	Start	Login	Reg.	About	Dashb.	Logout	Publ.	Profile	Nearby	Like
Start	0.2	0.4	0.3	0.1	0	0	0	0	0	0
Login	0	0.2	0.1	0	0.7	0	0	0	0	0
Register	0	0.1	0.2	0.7	0	0	0	0	0	0
About	0.6	0	0	0.4	0	0	0	0	0	0
Dashb.	0	0	0	0	0.2	0.1	0.1	0.1	0.5	0
Logout	0.8	0	0	0	0	0.2	0	0	0	0
Publish	0	0	0	0	0.7	0	0.3	0	0	0
Profile	0	0	0	0	0	0	0	0.3	0.7	0
Nearby	0	0	0	0	0.2	0	0	0.1	0.3	0.4
Like	0	0	0	0	0	0.2	0	0	0.7	0.1

Table 7.1: Predicted transition matrix of user-behaviour before implementation in Sprint 0.

For calculating the limiting distribution there are two ways: (1) Solving the balance equations, or (2) calculating the power of the transition matrix till an equilibrium is reached.
In order to give the steady state of the system a meaning in terms of arrivals to each state, two parameters need to be defined:

1. Estimated arrival-rate to the Travelistr-system

2. Estimated time a user on average spends in the Travelistr-system per interaction

The attentive reader will not be surprised that these parameters are applied using Little's Law to calculate the average number of users in the Travelistr-system. The application of Little's Law is possible because the system depicted in Figure 7.1 is ergodic.
For the initial evaluation an estimated arrival-rate of 2 users per second and a mean session-duration of 15 seconds is assumed.
Applying Little's-law, by simply multiplying the arrival-rate with the average session duration results in an average of 30 active users in the Travelistr-system.
The next step is to calculate the average number of users in each state and based on that the transition-rates between states can be calculated (Equations 6.2 and 6.3). The last step is to sum the incoming transitions of each state (Equation 6.4) and then take the inverse to retrieve the time between two consecutive arrivals, the inter-arrival rate. A more detailed description of the transformation approach from Markov models to queuing networks, or in fact arrival rates, can be found in Section 6.7. In Table 7.2 the results of these calculations are presented. As can be seen, the states *Nearby*, *Like* and *Dashboard* have the biggest fractions. This is not surprising, as these are assumed to be the main states/Features of Travelistr.

P	Steady state	E(N)	λ per sec	$1/\lambda$
Start	0.106	3.18	2.562	0.39
Login	0.059	1.77	1.413	0.71
Register	0.047	1.41	1.131	0.88
About	0.073	2.19	0.318	3.14
Dashboard	0.145	4.35	3.486	0.29
Logout	0.052	1.56	0.435	2.30
Publish	0.021	0.63	0.435	0.75
Profile	0.064	1.92	1.388	0.14
Nearby	0.301	9.03	6.909	0.28
Like	0.134	4.02	3.612	0.28

Table 7.2: Results of steady state analysis and derived arrival rates to the Travelistr system given predictive knowledge in Sprint 0.

7.5.1 Operations and service times on resources

After calculating the arrival-rates it is now time to define the operations that each state requires and the service-demands these operations result in. The resources of the Travelistr software system here are the CPU of the application-server, the database and the image-server.

Taking every single aspect of Travelistr into account may result in a big overhead and in general, a model never takes every aspect of the real system into account. This also is true for this model-based analysis here and also true for other performance evaluation frameworks and performance modelling standards. The CSM-model [62] for example requires the definition of »*Core*« - scenarios in a deterministic step by step manner.
However, more precise results that also take things like the implementation-style, the used frameworks and many others will be retrieved in Sprints 1, 2 and 3 by running empiric evaluations and load-tests.
As an approximation, the *handleRequest()* - operation is defined for every operation.
The states *Start*, *About*, *Logout* and *Dashboard* are mostly simple static web pages with no business-logic attached. Their only server-side logic is the handling of requests.

The six distinct core-operations of Travelistr are shown in Table 7.3. An operation in the nomenclature of the ASPE-process represents a class in a multiclass-queuing network. Extending the service times is, at least for me and I have more than five years of experience as a Software Engineer, very difficult however.

Given this simple approximative setup, it is obvious that the product-form assumption of queuing networks is not violated here and a wide range of queuing network solution-techniques can be used.
Here, the Mean Value Analysis (MVA) that calculates mean queue-lengths, mean utilizations and other mean values of the system is used. This also coincides with the fact that the transformation approach from Markov models to queuing models outlined in

Operation	λ	DB	CPU	ImageServer
handleRequest()	21.639	0	500	0
getUser()	3.882	750	500	0
saveUser()	1.131	900	500	0
publishImage()	0.435	600	2000	1000
getNearby()	6.909	1500	500	500
doLike()	3.612	700	500	0

Table 7.3: Distinct set of Travelistr operations and their service times in milliseconds predicted before implementation in Sprint 0.

Section 6.7 also primarily relies on mean values.

As an exmample for an immediate result of this calculation it can be easily calculated that on average 1566 images are uploaded to the Travelistr-system per hour. This is an interesting finding, as the effort of uploading, scaling and storing images is high. Furthermore, prizes for storing such an amount of pictures can be calculated when using a third party vendor.

7.5.2 Mean Value Analsyis

Given the results of Table 7.3, the mean value analysis can be carried out. To simplify this process, JMVA [30], a component of the Java Modelling Tool (JMT) is used. Please note that JMVA works on a seconds-basis, therefore, the input must also be in seconds. The main purpose of the mean value analysis here is to determine the *utilization* of resources. Please consider that usually *utilization* is regarded with values between 0 and 1 as it is defined as the percentage a station is utilized. However, here and throughout this work, *utilization* is regarded as the mean-value of the number of customers in a station. Therefore it can be a value bigger than one.
The basic setup has the following utilization shown in Table 7.4.

*	Aggreg.	handlReq.	getUser	saveUser	publImg.	getNearby	doLike
DB	17.0823	0.0	2.9115	1.0179	0.261	10.3635	2.5284
CPU	19.4565	10.8195	1.941	0.5655	0.87	3.4545	1.806
ImageS.	3.8895	0.0	0.0	0.0	0.435	3.4545	0.0

Table 7.4: Resource-utilization: Average number of users in each station DB, CPU and ImageServer in the Travelistr system calculated with predictive knowledge in Sprint 0.

These figures show a high utilization of CPU and DB. Especially for Travelistrs main feature *Nearby*, there is a very high utilization at the database. This result is very helpful as it gives a very good first overview of the resource utilization. One might already adapt the architecture or maybe scale the computing resources for CPU and DB horizontally-/vertically.

However, all of the data is predictive and done somehow with a *guts*-feeling. It is now very obvious that practitioners seek for measured data. And that is the reason I conclude *Sprint 0* here and continue with the implementation of Travelistr in *Sprint 1*.

7.6 Sprint 1

After setting up the environment and analysing the Travelistr-software system in a predictive way in *Sprint 0*, it is now the time to start developing. At the end of the Sprint, measurement-data from the deployed software system is retrieved.

Following Travelistr-features/pages are developed in *Sprint 1*: (1) *Start*, (2) *About*, (3) *Register*, (4) *Login* and (5) *Profile*. The necessary respective operations are (1) *handleRequest()*, (2) *getUser()* and (3) *saveUser()*.
OperationTraceMonitor-insertions are added to these operations. Additionally all interfaces- and operation- stubs are defined as well in order to easily launch full load-tests.

7.6.1 Running load tests

Two important aspects for combining the model-based- and the measurement-based approach with the ASPE-process is to *force automation* and *reuse artefacts*. The user-behaviour model defined in Markov-model notation is therefore reused in the Markov4JMeter-tool [79], an extension of JMeter.
A *Markov Session Controller* has to be created within a *Thread Group*. Sub-elements of the Markov Session Controller are the Markov states. The transitions between the states are defined with *guards* and *actions*. After that, the *User Behaviour Models* have to be defined. As outlined in the previous chapter only a single user-behaviour-model is used in this approach if there is no reason against doing so. The user-behaviour-model is stored as a CSV-file and simply contains the structure of the Markov-model transition matrix in Table 7.1. But here an important difference has to be outlined: In the ASPE-approach, the think-times are expressed by a states transition probability to itself. Defining it like this in Markov4JMeter would result in additional calls of that state. Think-times in Markov4JMeter are in contrast described with distribution-functions such as a Gaussian-distribution.
The manual creation of this probabilistic Load-test model with Markov4JMeter is quite a lot of work and furthermore error-prone. Automatic transformations from Markov-model descriptions towards load test plans would be highly valuable. However, a very useful side effect of using load-tests is that they can always also be seen as integration tests.
Hypothesis 5 in the previous Chapter is, that measurements on only language level are sufficient to estimate resource utilizations. In order to not cause delays and falsify the measurements, only one user is simulated with a waiting time of around one second after each request call. Besides the software system is not being used by others. This ensures that the data is not corrupted by contention and good approximations can be derived from the observations.

Limitations of Markov4JMeter
Markov4JMeter has useful concepts. However, a big problem that makes it unusable for Travelistr is that in Markov4JMeter a start-state must be defined without having the possibility to define other entry points nor how long a user stays in a system.
This results in a situation where the starting state is entered after every loop and a loop only reaches a couple of other states before terminating. The consequence is a very high utilization of the starting state and its neighbours. But states *more far* away from the starting state are visited very rarely.
The distribution of state visits is therefore not converging to the steady state solution.
Especially for modern Web 2.0 software systems it is assumed that the behaviour of Markov4JMeter is not appropriate. The session information (especially for mobile apps) is not deleted for very long times, or often is never deleted so that the user does not enter the starting state again in a long time. The resulting resource utilization of modern software systems are therefore assumed to be closer to a systems steady state solution (see Hypothesis 4).

7.6.2 Retrieving service times

The focus now lies on measuring the service times of operations on resources. In the JMeter *View Results Tree*, all sent http-requests- and responses are listed. Also it is possible to extract graphs for different performance metrics using available plugins. However, as also the user-behaviour needs to be tracked in *Sprint 3* in an empiric evaluation with real users and in order to evaluate Hypothesis 5, instead of using JMeter, the self-developed *OperationTraceMonitor* is integrated in the Travelistr software system. Furthermore, it is necessary to retrieve the response times of operations which would anyways not be possible with JMeter.

As outlined, the relevant operations in Sprint 1 are *handleRequest()*, *getUser()* and *saveUser()*. These operations together with a Java Servlet-Filter measuring begin and end of a request where instrumented with *OperationsAndTraceMonitor*-probes.
The *OperationsAndTraceMonitor* produces two CSV-files: (1) operationDuration.csv and (2) userTrace.csv.
The *operationDuration*-file has the following attributes: *operationID*, *startTime*, *endTime*, *duration*, *input*, *output* and *error*.
The *userTrace*-file is used in this work only for generating the user-behaviour Markov model and is therefore described in the respective Section 7.8.1. The load test described in the previous section resulted in an *operationDuration.csv*-file that is analysed with R-software [64] as follows.

Operation *getUser()* service times:
After removing outliers, a histogram showing the service times of the *getUser()*-operation on the database is depicted in Figure 7.2.

As can be clearly seen, there are two accumulation points. The main question now is, why is this the case?

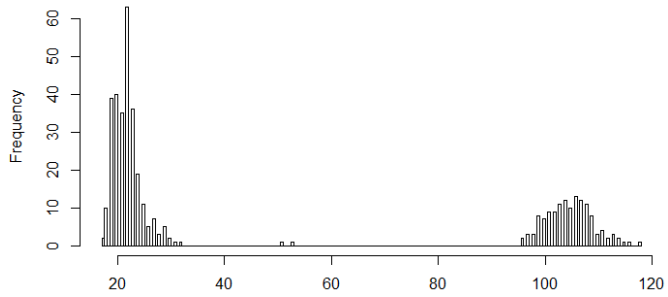

Figure 7.2: *getUser()* - service times in ms measured on the deployed Travelistr-system after Sprint 1, n = 416.

The first assumption was, that this is a result of the following: The *getUser()*-operation is used in combination with *saveUser()* in the *Registration*-feature. However, if someone registers, usually there is no entry of this person in the database whereas *getUser()* returns null. These requests are faster then those for the Login-feature, where most of the time, a user-object is returned.
However, this assumption could not be confirmed as there is no difference with statistical significance between requests that return no user-object and those who do.
The second assumption was that this is a result of caching. And indeed, login attempts for users that were shortly before retrieved were faster than the others, as they still were in the Cache of the used persistence-provider framework Hibernate on the application-server and no request to the database was made. But also removing this still resulted in two accumulation-points.
The third assumption, which turned out to hold, was that the used connection-pool c3p0 influenced the results. The overhead for creating a connection is quite big and this also influences the service times. The solution was to get the connection before starting the time measurement. And for the service-times, these setup-costs of the connection-pool are not regarded as we are interested in the steady state here. And in the steady state with a correct connection-pool setup, connections don't need to be created as they are reused and the reconnection is not regarded here.

A conclusion therefore is, that for analysing the performance it is also necessary to first be very careful where and how the service-times are retrieved and second which technologies are being used as they may influence the performance. A statistical evaluation as done here showed effectiveness to find such effects in data.
After correcting the test-setup and randomizing the Login-requests, the data had the form as shown in Figure 7.3. The arithmetic mean for the *getUser()*-operation is 18.02 ms.

Operation *saveUser()* service times:
The observations of the *saveUser()*-operation are shown in Figure 7.4. The arithmetic mean of the *saveUser()*-operation is 53.29 ms.

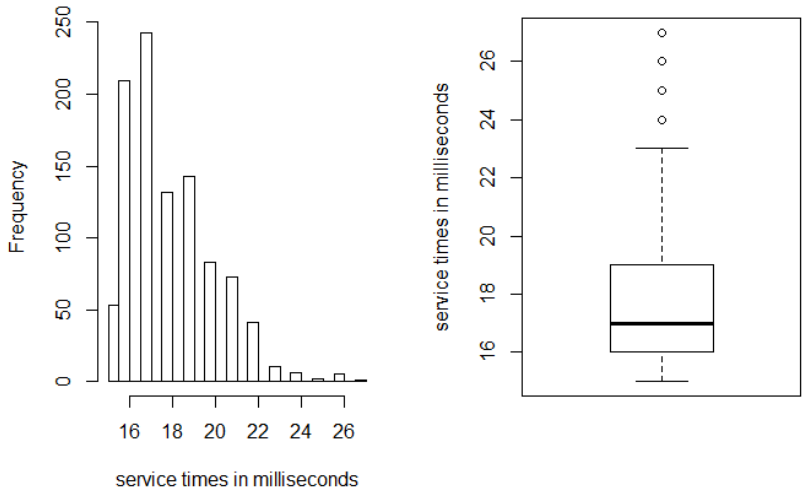

Figure 7.3: Service times of operation *getUser()* measured on the deployed Travelistr-system after Sprint 1, n = 1001.

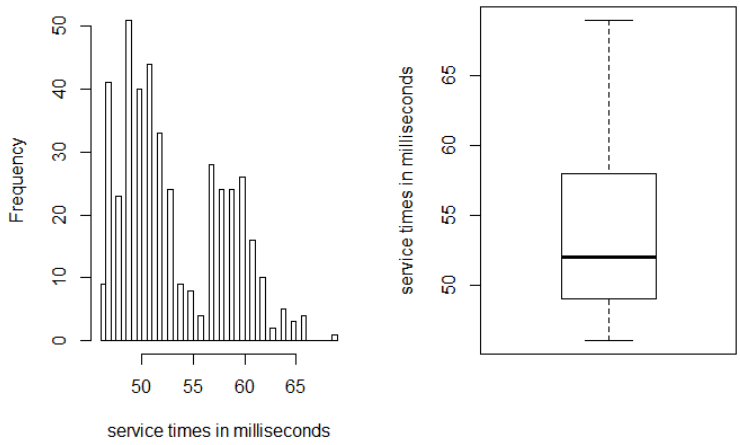

Figure 7.4: Service times of operation *saveUser()* measured on the deployed Travelistr-system after Sprint 1, n = 429.

Operation *handleRequest()* service times:
The handleRequest() service time here is defined as the difference between response-times and summed operation service times. The response-times show a couple of accumulation points and a very broad variance. That is not surprising as the GET-request for the Start-page does not require any operation other than handling the request. In contrast for example, register needs the two operation *getUser()* and *saveUser()*. So

there are two assumptions tested as follows: (1) response times are dependent on the requested page/feature and (2) the *handleRequest()*-operation is almost independent on the requested page.
If the second assumption doesn't hold, new operations must be defined that replace the default *handleRequest()*-operation. In Figure 7.5 the response-times for the different pages is shown.

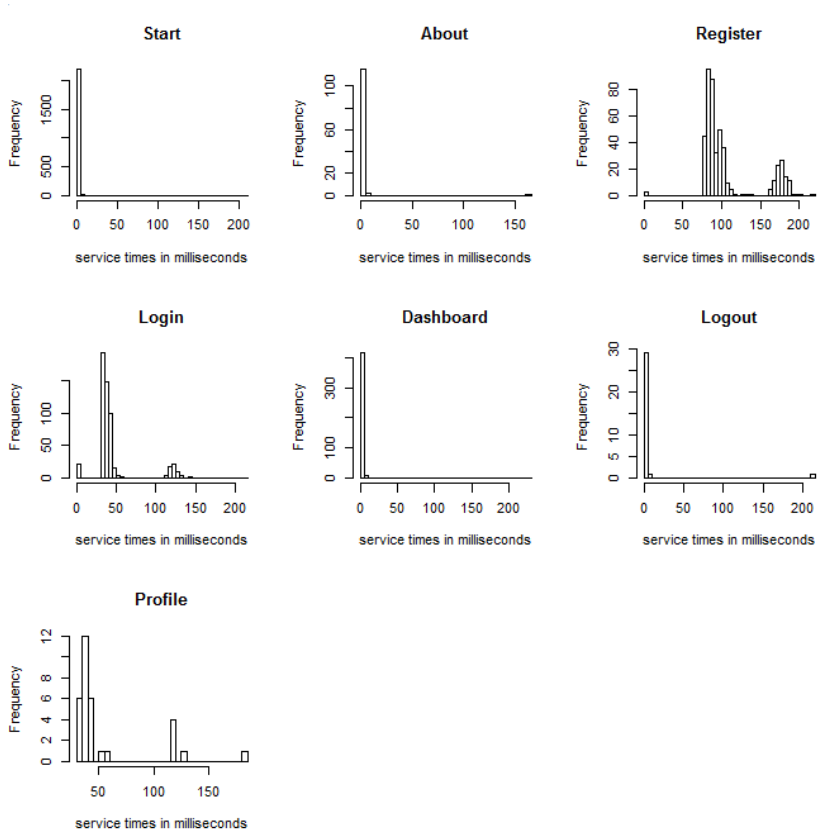

Figure 7.5: Response times of the different already implemented pages at the end of Sprint 1.

As can be seen, there is a difference between dynamic-(*Register*, *Login* and *Profile*) and static (*Start*, *About*, *Dashboard*, *Logout*) pages. The phenomena previously described for the *getUser()*-operation of accumulation points can also be seen here. However, to some extent, this is a characteristic of the software system at this level and therefore has to be regarded as such. The arithmetic mean-values of the response-times are: *Start*: 1.15 ms, *About*: 3.07 ms, *Register*: 107.54 ms, *Login*: 46.48 ms, *Dashboard*: 2.18 ms, *Logout*: 8.71 ms and *Profile*: 56.56 ms.
As an approximation of the *handleRequest()*-service-time of the respective pages, their respective sum of service times are subtracted from their average response times. The result is, that every page has a different mean value and due to this differences the

handleRequest()-operation is split up for every page. The summarized results of the analysis of observed service times after Sprint 1 can be found in Table 7.5.

Operation	λ	DB	CPU	ImageServer
handleRequestStart()	2.562	0	1.15	0
handleRequestAbout()	0.318	0	3.07	0
handleRequestLogin()	1.413	0	28.46	0
handleRequestRegister()	1.131	0	36.23	0
handleRequestDashboard()	3.486	0	2.18	0
handleRequestProfile()	1.388	0	38.54	0
getUser()	3.882	18.02	0	0
saveUser()	1.131	53.29	0	0

Table 7.5: Travelistr operations and their service times in milliseconds observed on the deployed software system at the end of Sprint 1.

7.6.3 Observing user-behaviour

Observing the user-behaviour after each Sprint is not part of this work. It appears to be of little sense to launch user-tests after the first Sprint as too little features are available. However, an interesting research direction for future work is, if observed user-interactions on a limited set of features can be extrapolated to a systems entire user-behaviour model in some way.

7.6.4 Enriching the MVA-model

The parameters of the MVA analysis for the entire Travelistr software system are enriched with the measurement-data derived in this Sprint and outlined in Table 7.5. The results of this analysis in terms of utilization are as follows.

- DB: 13.28
- CPU: 6.28
- ImageServer: 3.89

These figures describe the average number of users in each station DB, CPU and ImageServer. As can be seen, there is a strong decrease in the expected utilization for all three resources.

7.7 Sprint 2

In Sprint 2, only one feature is implemented: The upload of an image, *publishImage()*. Images are stored at the third-party cloud-vendor Cloudinary [5]. In order to have more

fine-grained information on the necessary steps to uploading a picture, the *publishImage()*-operation has four defined sub-operations: (1) *uploadImage()*, (2) *scaleImage()*, (3) *uploadToStore()* and (4) *saveImageRef()*.
The collected data for this sprint is also structured based on these sub-operations. The service-times of all these operations (except *saveImageRef()*) are affected by the size of the image that is uploaded. The Travelistr system allows its users to upload pictures up to a size of 10MB. It is assumed that most people use Travelistr with their mobile phones whereas 1MB seems to be a good guess for the average size of pictures uploaded. The test was therefore launched exclusively uploading a 1MB picture many times.
Figure 7.6 shows the observed service-times for the four operations.

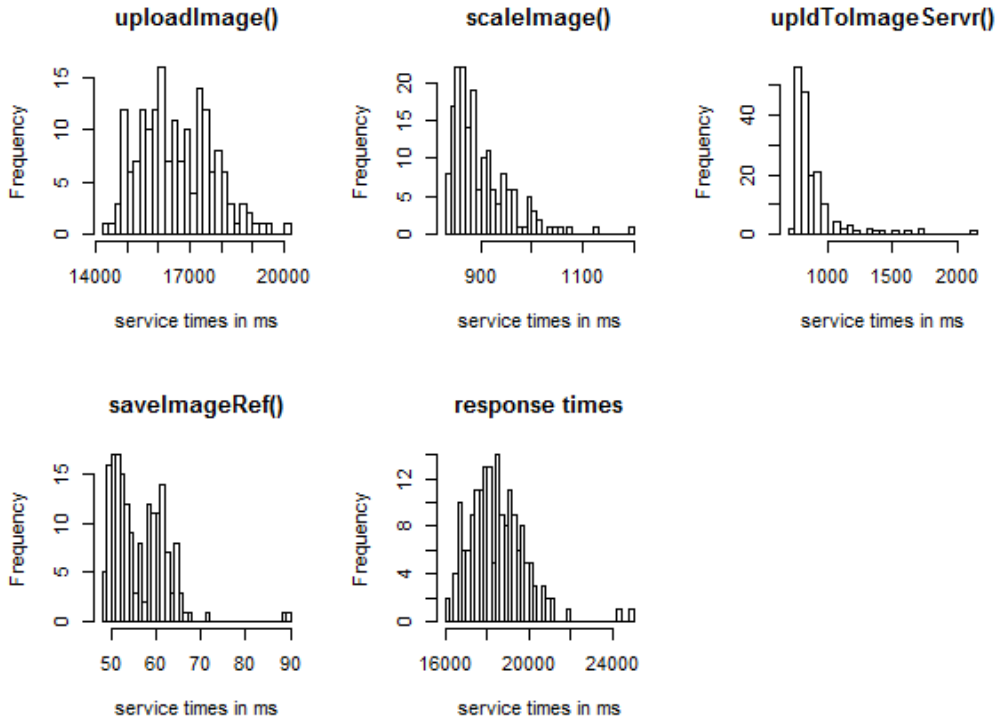

Figure 7.6: Response times of the sub-operations of *publishImage()* and the total response times, n = 176.

As can be seen, the *uploadImage()*-operation takes very long, on average roughly 16.5 seconds. When running this load-test, i was in my home village in a very rural area of Austria with a very bad internet speed. Launching this test again in my flat in Vienna shortens the time to about 4 seconds. And when I carried out the test in the WIFI at the campus of the Vienna University of Technology it completes after about 2 seconds. This shows the high dependence of service-times lengths on the infrastructure of a user for the *publishImage()*-feature.
It is very clear that with my approach of only measuring service-times at language level,

it is not possible to extract accurate service times for the *uploadImage()*-operation. The problem is, that the transmission does not depend on *own* hardware but also heavily on the users-hardware. This can only be mitigated when also the hardware-resources are instrumented in order to retrieve valid service-times. Due to this finding, Hypothesis 5 has to be rejected.
For now, due to the lack of a accurate measurement data, the *uploadImage()*-operation is changed in a predictive way to 1.5 seconds on the CPU which also includes the time to scale the image.
The mean values of the other operations are: *scaleImage()*: 900.01 ms, *uploadToImageServer()*: 889.26 ms and *saveImageRef()*: 56.83 ms. Furthermore the *handleRequestPublish()*-operation takes 70.8 ms.

The service-times of operations developed and analysed in Sprint 1 have not changed significantly and therefore their service times are kept as they were. In Table 7.6 the updated list of measured service-times is shown.

Operation	λ	DB	CPU	ImageServer
handleRequestStart()	2.562	0	1.15	0
handleRequestAbout()	0.318	0	3.07	0
handleRequestLogin()	1.413	0	28.46	0
handleRequestRegister()	1.131	0	36.23	0
handleRequestDashboard()	3.486	0	2.18	0
handleRequestProfile()	1.388	0	38.54	0
getUser()	3.882	18.02	0	0
saveUser()	1.131	53.29	0	0
publishImage()	0.435	56.83	1500	889.26

Table 7.6: Distinct set of measured Travelistr operations of the deployed system after Sprint 2 and their service times in milliseconds.

7.7.1 Enriching the MVA-model

The parameters of the MVA analysis for the entire Travelistr software system are enriched with the measurement-data derived in this Sprint and outlined in Table 7.6. The results of this analysis in terms of average utilization (number of customers at each station) are as follows. DB: 13.05, CPU: 6.06 and ImageServer: 3.84.
Again, a decrease in the expected utilization can be seen compared to the results of *Sprint 2*.

7.7.2 Retrieving the order of operation-execution

As outlined in Chapter 5, an information-requirement for many intermediate-format notations such as PMIF, CSM or KLAPER is the ordering of operations.
As these models are all in the domain of model-based performance engineering, the

ordering of operations is defined in a predictive way before implementation. However, a useful automated tool may retrieve the orderings and also the probability of operation-executions from observation data.
The research question herefore is as follows: *How can orderings of operations as well as execution-probabilities be derived from observation data?*

7.8 Sprint 3

Sprint 3 is the final iteratio for developing Travelistr. The two features *Nearby* and *Like* were developed and the profile-page was extended. Additionally the connection handling was adapted. Instead of using separate connections for every operation, the »*Connection per Thread*«-pattern was introduced. The connection-pooling was also adapted to the expected amount of users in a sense that at least 50 connections are always kept open and are reused and shared between users. Furthermore, also the passwords are hashed server side and a *Salt* is added.

The respective operations added in this Sprint are *getNearby()* and *doLike()*. The operation *getNearby()* was split into two sub-operations *getRecentTrending()* and *orderRecentTrendingByDistance()*.
An interesting finding here is the fact that now there is a significant difference in the *getUser()*-operation duration when called from the *Register*- and *Login*-feature. The reason for this is that together with the User-object, also it's pictures are loaded eagerly resulting in the *Login*-feature having a bigger response time than the *Register*-feature. Also, the *getUser()*-operation is split into two operations: *getUserEmpty()* and *getUserExists()*. When compared to the results of *Sprint 2*, the results show a high impact of the changes.

After that, Travelistr is in the Beta-status. All its features are available. Again, synthetic load tests with JMeter were launched. The final results of operation durations can be found in Table 7.7. The result of the MVA-analysis is as follows. DB: 2.33, CPU: 0.69 and ImageServer: 0.39.
Just like in Sprint 1 and 2, the expected utilization decreased for all three resources.

In Figure 7.7, the results of the MVA-analysis for Sprint 1, 2 and 3 and the respective resources DB, CPU and ImageServer is shown. The final numbers including nearly complete measurement data is far away from the predictions in the beginning. This is due to high predictions in the first place but also due to performance improvements such as enhanced connection-pooling. Additionally I assume that also for practitioners it is hard to predict operation-durations before development. As shown in Figure reffig:utilizationView for every Sprint, the expected utilization of resources dropped.
As outlined earlier, no adaptation of the expected user-behaviour was done during development. To replace the predicted user-behaviour by measured user-behaviour has not been done so far and is subject the next section.
Based on the results of Sprint 3, an adaptation of the computing capabilities of the Travelistr-software system seems necessary. For the empiric user-test, an amount of about

Operation	λ	DB	CPU	ImageServer
handleRequestStart()	2.562	0	3.33	0
handleRequestAbout()	0.318	0	2.88	0
handleRequestLogin()	1.413	0	1.41	0
handleRequestRegister()	1.131	0	2.05	0
handleRequestDashboard()	3.486	0	2.18	0
handleRequestProfile()	1.388	0	8.09	0
getUserEmpty()	1.131	88.38	0	0
getUserExists()	2.801	158.78	0	0
saveUser()	1.131	52.52	0	0
publishImage()	0.435	56.83	1500	889.26
getNearby()	6.909	199.26	0.1	0
doLike()	3.612	89.047	1.47	0

Table 7.7: Distinct set of measured Travelistr operations of the deployed system and their service times in milliseconds after Sprint 3.

30 users maximum is expected just as the calculations shown above. Given the results the only adaptation is an increase of computing capabilities at the DB. The way this optimization and increase is handled is not described here. It is not subject of this work and there is a tremendous amount of literature to this topic available (such as [46]).

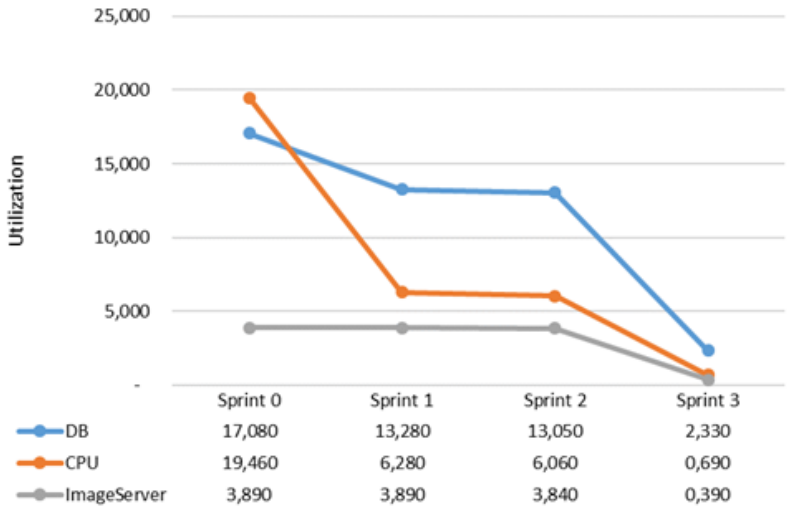

Figure 7.7: Expected Utilization of the resources of the Travelistr software system over time at the end of the development Sprints.

7.8.1 Evaluating the Markov assumption for Travelistr

The purpose of this section is to evaluate Hypothesis 2 with the observed user traces of the empiric user test that was carried out using the Travelistr system. To do so, an approximative approach proposed by Li et al. [55] is applied. Here, transition-probabilities of a Markov model of order one are compared to transition-probabilities of order two. If the probabilities of order one and two don't diverge more than a certain threshold, the Markov assumption is accepted.

The observational data is retrieved using the *UserTrace*-component of the *OperationsAndTraceMonitor*-tool. User traces here are stored in a CSV-file. The retrieval of a Markov model given the data in a CSV-format is a sophisticated error-prone task. Such tasks are ideally automated by using a respective software tool. The software tool implemented therefore is called *userTrace2Markov* and is available online [8]. It is a command-line tool written in JAVA that accepts four parameters. The first parameter is the path to the input file, the second parameter the path to the output file, the third parameter describes how long a user interaction is considered as one interaction and the fourth (optional) parameter is the name of the software system. For this evaluation, user-interactions are considered as cohesive if two consecutive requests are within 120 seconds.

The input file must be a CSV-file and include the following attributes: *entered-path*, *time*, *userIP*, *sessionID*, *userID*, *action*.
The action can in the first version of the *UserTrace2Markov*-tool only be of type *GET*.
The results of the analysis are written to a file defined in the second parameter in form of a txt-file. The tool calculates transition-rates between states for a first- and second order Markov chain. Furthermore, think-times of every state are calculated and described by mean-value and deviation of a standard normal distribution. Additionally, the total aggregated time spent in each state by every user is summed up. This is useful for analysing the steady state assumption of Hypothesis 4.

The results of the empirical user-test in form of a first-order Markov model are shown in Figure 7.8.

In the appendix, the detailed results for every state and every transition can be found. The states *Like* and *Liked* were omitted for evaluation because these pages had simple redirects, resulting in no usable information gain for my purposes. Furthermore also the states *Logout* and *About* were omitted as they do not provide the users with sufficient freedom of decision where to go next. An evaluation would therefore also result in no information gain.

The user test took place for one week. In total, 32 users participated. All of them are my personal friends and I asked them to join the test in the following way:

»*For the practical part of my master thesis I do an empiric test. Right now, I am looking for some test-users. Would be cool if you participate also. The Travelistr-platform is very simple. All you need to do is register and from time to time upload a picture or like other pictures. Comparable to how you would use Instagram for example. The test will run till*

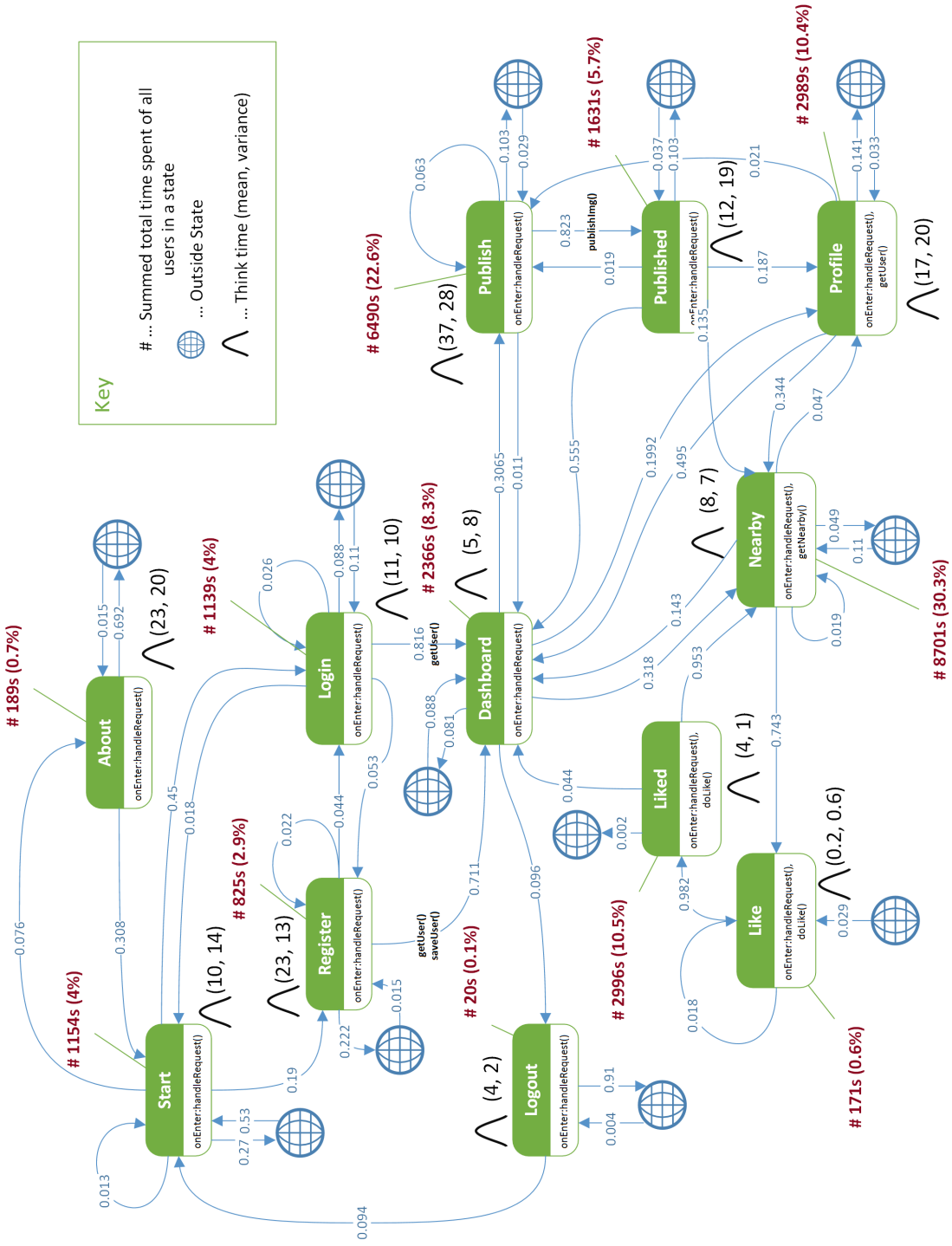

Figure 7.8: Results of the empirical user test in form of a first order Markov model.

the end of October.
The test is about evaluating the Markov-assumption for the usage-profile of websites in the domain of Software Performance Engineering. A more detailed description about Travelistr and more background information can be found directly at the welcome-page of Travelistr.
Your usage data will only be used anonymously!
Here is the link to Travelistr:
`http://138.68.73.16:8080/TravelistrApp/start`
To participate you only need to register (please remember the password) and then you can already use all features of Travelistr. You can start right away as the test is already running.
Thanks in advance!«

146 pictures were uploaded to Travelistr which received 839 *Likes* altogether. 4520 transitions between states were observed within 173 distinct user interactions. Two consecutive transactions from a user are considered to be in the same user interaction if their timely difference is equal or less than 120 seconds. To distinguish users, a randomly generated ID was generated for each session and, besides being stored server side, also copied to the users local machine with the use of Cookies. A user-session was never invalidated during the whole test. So if a user hasn't on purpose logged out, the user could continue were he/she left the last time without logging in again. The results for the approximative evaluation of the Markov assumption can be found in Table 7.8.

State	Avg. information loss from 2nd to 1st order model	Observed Transitions (n)
Nearby	4.36%	1067
Dashboard	11.45%	506
Profile	6.72%	193
Publish	2.87%	190
Published	1.77%	153
Start	7.58%	128
Login	3.14%	106
Register	12.48%	35

Table 7.8: Results of comparing the first-order Markov model solution of user-behaviour with the second-order Markov model.

These results confirm Hypothesis 2. Markov models of order one seem to be good approximations of user-behaviour of typical Web 2.0 applications. However, the artificial user test carried out has certain threads to validity that are described in Section 7.10. Furthermore, in Chapter 8, future research directions and remaining research questions are outlined that build upon this works results.

7.8.2 Empirical results compared to the steady state

Hypothesis 4 states that the distributions of the summed total time users on average spent in each state will converge to a steady state solution of the Markov model. In Table 7.9, the total time users spent in each state is outlined.

State	Total time (in sec)	Distribution	Probability to enter from Outside
Nearby	8701	30.3%	11.0%
Publish	6490	22.6%	2.9%
Liked	2996	10.5%	0.0%
Profile	2989	10.4%	3.3%
Dashboard	2366	8.3%	8.8%
Published	1631	5.7%	3.7%
Start	1154	4.0%	53.0%
Login	1139	4.0%	11.0%
Register	825	2.9%	1.5%
About	189	0.7%	1.5%
Like	171	0.6%	2.9%
Logout	20	0.1%	0.4%

Table 7.9: Observed total times summed of each user that were spent in each state of the Travelistr system.

Hypothesis 3 states, that users in typical Web 2.0 applications are not bound to a specific starting state. As can be seen in Table 7.9, this is true for this evaluation. I furthermore assume, that given observations of longer periods of empiric studies, the probability of entering a state from outside will be somehow connected to the steady state distribution.

Hypothesis 4 cannot be accepted or rejected here. The Travelistr case study is also of exploratory nature. Here, additional parameters were identified that were not taken into account in the first place. These new parameters that may need to be taken into account are:

- Specific probabilities for entering specific states.
- Self-references of states (e.g. a refresh on a website). Because of that, the transition matrix has to be defined in a different way and also the think-times have to be regarded differently.

A new mathematical approach may be defined for this changed circumstances that also takes these new parameters into account. Given the observed data, the simple steady state calculation however is assumed to be a valid approximation but reliable assertions based on the given data is not possible. Hypothesis 4 is not accepted nor rejected

therefore. An empiric study evaluating already productive applications in future research is needed to evaluate this Hypothesis.

Assumption 1 is true for Travelistr and is assumed to be true for many other software systems. An important aspect of ergodic systems is, that there are no seperated states or islands of states that cannot be reached from others. Such software systems where features are unreachable given a specific state would make little sense I assume.

7.9 Benefits and Overhead

Taking performance early and throughout a software project into consideration showed high utility in the Travelistr case study.
The predictive model-based approach in SPE alone is not accurate as it does not take language-specific, implementation style and other factors into account. An example for this in the Travelistr case study is the use of a connection-pool that would be hard to describe predictively. Furthermore, changes in the implementation and the architecture can be compared to earlier versions and their effect is measurable. Furthermore, I assume that for practitioners it is nearly impossible to *guess* the operation durations and resulting resource utilizations beforehand. Therefore, measurements on deployed software systems are very useful.

However, there is overhead associated to the ASPE approach. Therefore, the existence of respective automation tools is essential. Given, only the need to install resource/language level probes and define a system in a way such as depicted in Figure 7.8 seems to be reasonable and usable for practitioners. Furthermore, as the domain is rather complex, these tools are able to hide away the complexity of the approach. Especially the use of order one Markov models is very easy compared to other approaches.

7.10 Threads to validity

The empiric user test evaluating the Markov assumption and other hypotheses was successful. However, 32 users took part in the test and not every state/transition was visited as often as it would be necessary. To reduce the risk of evaluating bad data, only states and their respective transitions with more than 20 observed transitions were taken into consideration for evaluating the Markov assumption.
Furthermore, there is a risk that the users did not use Travelistr as they would if Travelistr was a real application used in their daily lifes. To mitigate this problem, I only asked personal friends of mine i can trust they would keep on using it for a while in an appropriate way. A second initiative to mitigate this risk was to include a gamified approach where each night, an email with the *Most Liked Pictures* was sent out to all users together with the statistics for their performance on Travelistr with regard to how many *Likes* they got the day before.
The evaluation period is another risk, as it is assumed, that the behaviour of users over a longer period accordingly changes due to a better understanding of the system or due to

the simple fact that the registration only needs to be done once and never again. The fractions of some states (e.g.*Register*) is therefore assumed to be different in the long run. This is also a reason why Hypothesis 4 could not be evaluated in this work.

CHAPTER 8

Conclusion and Outlook

The proposed ASPE-approach combines model-based techniques and measurement-based techniques in SPE and showed utility in a case study for developing a typical Web 2.0 application.

Sub-goal one (SG1) was to propose a metamodel describing the information requirements for adequately evaluating a software systems performance. Herefore, the CETO-metamodel was defined which showed validity and effectiveness in the Travelistr case study.

The second sub-goal (SG2) for proposing an agile software engineering process that includes SPE in a seamless manner was achieved by outlining the ASPE-process. The agile process showed effectiveness for retrieving and defining static-, predictive- as well as measurement data. The validity and effectiveness of the ASPE-process was demonstrated in the Travelistr case study. Furthermore, its usability and utility for real world projects was shown. Sub-goal 2 was expressed in Hypothesis 1, which was the main research interest of this work. Respective automation tools are however necessary to be usable for practitioners.

Sub-goal three (SG3), an approach to retrieve queuing networks from Markov user-behaviour models, was achieved. For the outlined transformations, necessary conditions need to hold, especially ergodicity as well as the Markov assumption for user-behaviour. These conditions are expressed in Hypothesis 1 and 2. However, Hypothesis 4 could neither be accepted nor rejected in this work and needs to be evaluated in future research.

The fourth sub-goal (SG4) was the proposition of a metamodel for performance analysis that describes workloads on resources. Herefore, the MUPOM metamodel was proposed which uses the Markov model formalism for describing the usage profile. The MUPOM-metamodel showed validity and effectiveness in the Travelistr case study.

The fifth sub-goal (SG5) was achieved by doing an empirical user test. The Markov assumption was evaluated with an approximative approach that shows the information loss from a Markov model of order two compared to a Markov model of order one. The user test showed that Markov models are valid approximations for modelling user-behaviour of typical Web 2.0 applications.

Hypothesis 5 was rejected, as especially service times that depend on users resources, such as the upload of an image through a local network, can not be determined appropriately without also adding resource-probes.

8.1 Future research directions

The Travelistr case study was, besides being of confirmatory nature, also of exploratory nature. As the SPE discipline, and in consequence my own research work, cover a broad range of domains, it is not surprising that further contributions are necessary. Some possible research directions are outlined as follows.

One interesting aspect is the extrapolation of user-behaviour based on a limited set of available features after an iteration as described in Section 7.6.3. The research questions herefore are: *Can observed user-interactions on a limited set of features be extrapolated to a systems entire user-behaviour model? How can these measurements be validated?* The practical consequence would be, that already during development, usage patterns can be quantified and besides enhancing the accuracy of operation durations, also the user-behaviour model can be adapted and enriched with findings from measurements over time. A possible solution approach herefore might be to mock the behaviour of missing features.

Another problem outlined in the case study applies for operations where the infrastructure of users has a strong influence on the request-response-interval. As stated above, measured service times on a language level alone are not sufficient. Instead, also observations on physical resources have to be retrieved in order to estimate the service-times more precisely. An interesting tool for doing this is Kieker [68], that integrates both, language-level- and resource-level instrumentation. Evaluating Kieker and other resource-instrumentation-tools for their applicability in the ASPE-approach is another research direction.

As outlined in Section 7.7.2, the retrieval of causal operation-orderings from observation data is a possible way for automating big parts of the performance engineering process. This research direction is suited around the following question: *How can orderings of operations as well as execution-probabilities be derived from observation data?* The practical consequence is a decreased overhead leading to higher utility.

An assumption and necessary pre-condition for using the ASPE-approach for developing a software system and managing its performance is, that the user-behaviour as well as the structure of the system in terms of navigability must be ergodic. This is expressed in Assumption 1. For Travelistr, which is a typical Web 2.0 application and the basis of the outlined case study, this assumption holds. However, also other types of software systems

may be of the same nature. The respective research questions are: *Are software systems likely to be of ergodic nature? What types of software systems are likely to be ergodic?*

As outlined in Section 7.9, tool support and automation is important for the ASPE-approach to be usable in practice. Therefore, a tool that takes information defined as in Figure 7.8 might be developed. This can either be done in a self-contained manner or via interfaces to other tools.

A further research direction is concerned with the concept of Elastic-Clouds, where computing power can be extended on demand. The research question here is as follows: *How can findings in the ASPE-approach be integrated into Elastic-Cloud frameworks? How can respective metrics and rules be derived from measurements? How can such rules be defined in a model-theoretic way?*

As outlined in Section 5.6, Hidden Markov Models seem to be a useful technique in SPE. A brief concept herefore may look as follows: Operation calls and use-case calls as well as the resulting user-behaviour model are retrieved by only observing resource-utilizations. From there, the underlying Markov process is derived using respective techniques. This might be especially useful if, due to some circumstances, emissions on a use case level can not directly be observed.

Acronyms

SPE Software Performance Engineering

ASPE Agile Software Performance Engineering

DevOps Development and Operations

SLA Service Level Agreement

QoS Quality of Service

SCRUM An agile software development framework

QA Quality assurance

Cloud Internet-based computing with shared resources

HMM Hidden Markov models

QN Queuing network

LQN Layered Queuing network

EQN Extended Queuing network

CPU Central Processing Unit

DB Database

MDE Model driven engineering

MDA Model driven architecture

OMG Object Management Group

MOF Meta Object Facility

IaaS Infrastructure as a Service

PaaS Platform as a Service

SaaS Software as a Service

OASIS Organization for the Advancement of Structured Information Standards

TOSCA Topology and Orchestration Specification for Cloud Applications

CSM Core Scenario Model

UML Unified Modeling Language

UML MARTE UML Profile for Modeling and Analysis of Real-Time Embedded systems

UML SPT UML Profile for SPT Schedulability, Performance and Time

PMIF Performance Model Interchange Format

QNM Queuing Network Metamodel

KLAPER Kernel Language for Performance and Reliability

PCM Palladio Component model

PUMA Performance from Unified Model Analysis

DTMP Discrete time Markov processes

OWL-S Web Ontology Language for Web Services

ATL Atlas Transformation Language

JMT Java Modeling Tool

JMVA Java Mean Value Analysis

GNU GNU's not Unix

GPL General Public License

HPG Hypertext Probabilistic Grammar model

CETO Components Emission and Timely Observations

MUPOM Markov Usage Process and Operations Measurements

Results of the empirical user test

Mean information losses

The results of the evaluation of each state are as follows:

State *Nearby*:

Number of transitions from or to this state (n) = 1067.
The average error of all transitions from an order 2 model compared to an order 1 model is: 4.36%.

Incoming state	**Transition Probability**	**Deviation**
Transition From *Nearby* to *Nearby*		
Dashboard	2.5%	0.6%
Liked	0.53%	1.37%
Published	0.0%	1.9%
Outside	7.7%	5.8%
Profile	3.08%	1.18%
~~Nearby~~	~~43.75%~~	~~41.85%~~
Overall	1.9%	**Avg. Error: 2.17%**
Transition From *Nearby* to *Like*		
Dashboard	65.20%	9.10%
Liked	81.64%	7.34%
Published	80.0%	5.70%
Outside	50.00%	24.30%
Profile	73.85%	0.45%
~~Nearby~~	~~12.5%~~	~~61.8%~~
Overall	74.3%	**Avg. Error: 9.38%**
Dashboard	21.52%	7.22%
Liked	12.43%	1.87%
Published	15.00%	0.70%
Outside	23.08%	8.78%
Profile	13.85%	0.45%
~~Nearby~~	~~31.25%~~	~~16.95%~~
Overall	14.3%	**Avg. Error: 3.80%**

From state	**Transition Probability**	**Deviation**
Transition From *Nearby* to *Profile*		
Dashboard	8.86%	4.16%
Liked	3.57%	1.13%
Published	0.00%	4.70%
Outside	11.54%	6.84%
Profile	7.7%	3.00%
~~Nearby~~	~~6.25%~~	~~1.55%~~
Overall	4.70%	**Avg. Error: 3.97%**
Transition From *Nearby* to *Outside*		
Dashboard	1.90%	3.00%
Liked	1.85%	3.05%
Published	5.0%	0.10%
Outside	7.70%	2.80%
Profile	1.54%	3.36%
~~Nearby~~	~~6.25%~~	~~1.35%~~
Overall	4.90%	**Avg. Error: 2.46%**

State *Dashboard*:
Number of transitions (n) = 506;
The average error of all transitions from an order 2 model compared to an order 1 model is: 11.45%.

Incoming state	**Transition Probability**	**Deviation**
Transition From *Dashboard* to *Profile*		
Liked	19.40%	0.52%
Profile	0.00%	19.92%
Nearby	38.20%	18.28%
Login	25.60%	5.68%
Register	12.50%	7.42%
Published	10.10%	9.82%
Outside	4.50%	15.42%
Overall	19.92%	**Avg. Error: 11.01%**
Transition From *Dashboard* to *Logout*		
Liked	2.80%	6.80%
Profile	13.80%	4.20%
Nearby	22.90%	13.30%
Login	3.50%	6.10%
Register	0.00%	9.60%
Published	0.00%	9.60%
Outside	4.50%	5.10%
Overall	9.60%	**Avg. Error: 7.81%**
Transition From *Dashboard* to *Outside*		
Liked	5.60%	2.50%
Profile	10.30%	2.20%
Nearby	6.90%	1.20%
Login	0.00%	8.10%
Register	3.10%	5.00%
Published	0.00%	8.10%
Outside	9.10%	1.00%
Overall	8.10%	**Avg. Error: 4.01%**

From state	**Transition Probability**	**Deviation**
Transition From *Dashboard* to *Nearby*		
Liked	44.40%	12.60%
Profile	31.00%	0.80%
Nearby	20.10%	11.70%
Login	62.80%	31.00%
Register	50.00%	18.20%
Published	10.10%	21.70%
Outside	9.10%	12.60%
Overall	31.80%	**Avg. Error: 16.96%**
Transition From *Dashboard* to *Publish*		
Liked	27.80%	2.85%
Profile	31.00%	0.35%
Nearby	11.81%	18.84%
Login	15.10%	15.55%
Register	50.00%	19.35%
Published	88.00%	57.35%
Outside	22.70%	7.95%
Overall	30.65%	**Avg. Error: 17.46%**

State *Publish*:
Number of transitions (n) = 190.
The average error of all transitions from an order 2 model compared to an order 1 model is: 2.87%.

Incoming state	Transition Probability	Deviation
Transition From *Publish* to *Outside*		
~~Published~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	7.75%	2.55%
~~Publish~~	~~0.00~~%	~~0.00~~%
~~Profile~~	~~0.00~~%	~~0.00~~%
Overall	10.03%	**Avg. Error: 2.55%**

From state	Transition Probability	Deviation
Transition From *Publish* to *Publish*		
~~Published~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	3.50%	2.80%
~~Publish~~	~~0.00~~%	~~0.00~~%
~~Profile~~	~~0.00~~%	~~0.00~~%
Overall	6.30%	**Avg. Error: 2.80%**

Transition From *Publish* to *Published*		
~~Published~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	88.03%	5.73%
~~Publish~~	~~0.00~~%	~~0.00~~%
~~Profile~~	~~0.00~~%	~~0.00~~%
Overall	82.30%	**Avg. Error: 5.73%**

Transition From *Publish* to *Dashboard*		
~~Published~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	0.70%	0.40%
~~Publish~~	~~0.00~~%	~~0.00~~%
~~Profile~~	~~0.00~~%	~~0.00~~%
Overall	1.10%	**Avg. Error: 0.40%**

State *Published*:
Number of transitions (n) = 153.
The average error of all transitions from an order 2 model compared to an order 1 model is: 1.77%.

Incoming state	Transition Probability	Deviation
Transition From *Published* to *Publish*		
Publish	2.17%	0.27%
~~Outside~~	~~0.00~~%	~~0.00~~%
Overall	1.90%	**Avg. Error: 0.27%**

Transition From *Published* to *Dashboard*		
Publish	58.70%	3.20%
~~Outside~~	~~0.00~~%	~~0.00~~%
Overall	55.50%	**Avg. Error: 3.20%**

Transition From *Published* to *Nearby*		
Publish	14.50%	1.00%
~~Outside~~	~~0.00~~%	~~0.00~~%
Overall	13.50%	**Avg. Error: 1.00%**

From state	Transition Probability	Deviation
Transition From *Published* to *Profile*		
Publish	18.12%	0.58%
~~Outside~~	~~0.00~~%	~~0.00~~%
Overall	18.70%	**Avg. Error: 0.58%**

Transition From *Published* to *Outside*		
Publish	6.52%	3.78%
~~Outside~~	~~0.00~~%	~~0.00~~%
Overall	10.30%	**Avg. Error: 3.78%**

State *Profile*:
Number of transitions (n) = 193.
The average error of all transitions from an order 2 model compared to an order 1 model is: 6.72%.

Incoming state	Transition Probability	Deviation
Transition From *Profile* to *Publish*		
Nearby	0.00%	2.10%
Published	0.00%	2.10%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	2.22%	0.12%
Overall	2.10%	**Avg. Error: 1.44%**
Transition From *Profile* to *Nearby*		
Nearby	45.65%	11.25%
Published	55.17%	20.77%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	26.67%	7.73%
Overall	34.40%	**Avg. Error: 13.25%**

From state	Transition Probability	Deviation
Transition From *Profile* to *Outside*		
Nearby	4.35%	9.75%
Published	0.00%	14.10%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	14.44%	0.34%
Overall	14.1%	**Avg. Error: 8.06%**
Transition From *Profile* to *Dashboard*		
Nearby	50.00%	0.50%
Published	44.83%	4.67%
~~Outside~~	~~0.00~~%	~~0.00~~%
Dashboard	56.67%	7.17%
Overall	49.50%	**Avg. Error: 4.11%**

State *Login*:
Number of transitions (n) = 106.
The average error of all transitions from an order 2 model compared to an order 1 model is: 3.14%.

Incoming state	Transition Probability	Deviation
Transition from *Login* to *Register*		
~~Login~~	~~0.00~~%	~~0.00~~%
Start	5.97%	0.67%
~~Register~~	~~0.00~~%	~~0.00~~%
Outside	3.70%	1.60%
Overall	5.30%	**Avg. Error: 1.14%**
Transition From *Login* to *Dashboard*		
~~Login~~	~~0.00~~%	~~0.00~~%
Start	89.55%	7.95%
~~Register~~	~~0.00~~%	~~0.00~~%
Outside	85.19%	3.59%
Overall	81.60%	**Avg. Error: 5.77%**
Transition From *Login* to *Start*		
~~Login~~	~~0.00~~%	~~0.00~~%
Start	1.49%	0.31%
~~Register~~	~~0.00~~%	~~0.00~~%
Outside	3.70%	1.90%
Overall	1.80%	**Avg. Error: 1.11%**

From state	Transition Probability	Deviation
Transition From *Login* to *Outside*		
~~Login~~	~~0.00~~%	~~0.00~~%
Start	0.00%	8.80%
~~Register~~	~~0.00~~%	~~0.00~~%
Outside	3.70%	5.10%
Overall	8.80%	**Avg. Error: 6.95%**
Transition From *Login* to *Login*		
~~Login~~	~~0.00~~%	~~0.00~~%
Start	2.99%	0.39%
~~Register~~	~~0.00~~%	~~0.00~~%
Outside	3.70%	1.10%
Overall	2.60%	**Avg. Error: 0.75%**

State *Register*:
Number of transitions (n) = 35.
The average error of all transitions from an order 2 model compared to an order 1 model is: 12.48%.

From state	Transition Probability	Deviation
Transition From *Register* to *Dashboard*		
~~Login~~	~~0.00~~%	~~0.00~~%
Start	95.83%	24.73%
~~Register~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
Overall	71.10%	**Average Error: 24.73%**
Transition From *Register* to *Login*		
~~Login~~	~~0.00~~%	~~0.00~~%
Start	4.17%	0.23%
~~Register~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
Overall	4.40%	**Average Error: 0.23%**

From state	Transition Probability	Deviation
Transition From *Register* to *Register*		
~~Login~~	~~0.00~~%	~~0.00~~%
~~Start~~	~~0.00~~%	~~0.00~~%
~~Register~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
~~Overall~~	~~0.00~~%	~~**Average Error: 0.00%**~~
Transition From *Register* to *Outside*		
~~Login~~	~~0.00~~%	~~0.00~~%
~~Start~~	~~0.00~~%	~~0.00~~%
~~Register~~	~~0.00~~%	~~0.00~~%
~~Outside~~	~~0.00~~%	~~0.00~~%
~~Overall~~	~~0.00~~%	~~**Average Error: 0.00%**~~

State *Start*:
Number of transitions (n) = 128.
The average error of all transitions from an order 2 model compared to an order 1 model is: 7.58%.

Incoming state	Transition Probability	Deviation
Transition From *Start* to *Register*		
~~Start~~	~~0.00~~%	~~0.00~~%
~~Logout~~	~~0.00~~%	~~0.00~~%
Outside	24.32%	5.32%
~~About~~	~~0.00~~%	~~0.00~~%
~~Login~~	~~0.00~~%	~~0.00~~%
Overall	19.0%	**Avg. Error: 5.32%**
Transition From *Start* to *Start*		
~~Start~~	~~0.00~~%	~~0.00~~%
~~Logout~~	~~0.00~~%	~~0.00~~%
Outside	1.80%	0.50%
~~About~~	~~0.00~~%	~~0.00~~%
~~Login~~	~~0.00~~%	~~0.00~~%
Overall	1.30%	**Avg. Error: 0.50%**
Transition From *Start* to *Login*		
~~Start~~	~~0.00~~%	~~0.00~~%
~~Logout~~	~~0.00~~%	~~0.00~~%
Outside	57.66%	12.66%
~~About~~	~~0.00~~%	~~0.00~~%
~~Login~~	~~0.00~~%	~~0.00~~%
Overall	45.00%	**Avg. Error: 12.66%**

From state	Transition Probability	Deviation
Transition From *Start* to *About*		
~~Start~~	~~0.00~~%	~~0.00~~%
~~Logout~~	~~0.00~~%	~~0.00~~%
Outside	8.11%	0.51%
~~About~~	~~0.00~~%	~~0.00~~%
~~Login~~	~~0.00~~%	~~0.00~~%
Overall	7.60%	**Avg. Error: 0.51%**
Transition From *Start* to *Outside*		
~~Start~~	~~0.00~~%	~~0.00~~%
~~Logout~~	~~0.00~~%	~~0.00~~%
Outside	8.11%	18.89%
~~About~~	~~0.00~~%	~~0.00~~%
~~Login~~	~~0.00~~%	~~0.00~~%
Overall	27.00%	**Avg. Error: 18.89%**

Raw result data of the UserTrace2Markov-Tool

The results were retrieved with the *UserTrace2Markov*-Tool and are as follows:

```
Input: C:\Users\Johannes\OneDrive\TU_WIEN\MASTER\MasterThesis\Travelistr\Sprints\EmpiricalResults\
    sampleTrace.csv
Output: C:\Users\Johannes\OneDrive\TU_WIEN\MASTER\MasterThesis\Travelistr\Sprints\EmpiricalResults\
    results.txt
Duration: 120sec
Systemname: Travelistr
************************** General Information **********************************

System{name='Travelistr', states=[
State{id=''},
State{id='LOGOUT'},
State{id='REGISTER'},
State{id='LIKE'},
State{id='DASHBOARD'},
State{id='PUBLISHED'},
State{id='PROFILE'},
State{id='PUBLISH'},
State{id='LOGIN'},
State{id='NEARBY'},
State{id='START'},
State{id='OUTSIDE'},
State{id='LIKED'},
State{id='ABOUT'}]}
Altogether there were 4243 visits to states observed from 173 distinct users-interactions.
Total number of transitions: 4520
*********************************************************************************


Following the informations about the States:

State: State{id=''}
Think-time in ms (normal distributed):
Mean: 8166.666666666666
StandardDeviation: 8328.665359267754
Transition{from=State{id=''}, to=State{id='REGISTER'}} = 1
Transition{from=State{id=''}, to=State{id='NEARBY'}} = 1
Transition{from=State{id=''}, to=State{id='LOGIN'}} = 4
Total time of users spent in this state: 49sec
---------------------------------------------------------------------
Second order Markov model transitions:
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id=''}, next=State{id='NEARBY'},
    transition=1}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id=''}, next=State{id='REGISTER'},
    transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id=''}, next=State{id='LOGIN'},
    transition=3}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id=''}, next=State{id='LOGIN'},
    transition=1}
---------------------------------------------------------------------
State: State{id='LOGOUT'}
Think-time in ms (normal distributed):
Mean: 4000.0
StandardDeviation: 2121.3203435596424
Transition{from=State{id='LOGOUT'}, to=State{id='START'}} = 5
Transition{from=State{id='LOGOUT'}, to=State{id='OUTSIDE'}} = 48
Total time of users spent in this state: 20sec
---------------------------------------------------------------------
Second order Markov model transitions:
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='LOGOUT'}, next=State{id='OUTSIDE
    '}, transition=4}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='LOGOUT'}, next=State{id='START'},
    transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='LOGOUT'}, next=State{id='START'
    }, transition=3}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='LOGOUT'}, next=State{id='START'},
    transition=1}
---------------------------------------------------------------------
State: State{id='REGISTER'}
Think-time in ms (normal distributed):
```

```
    Mean: 23571.428571428572
61  StandardDeviation: 13346.701981445174
    Transition{from=State{id='REGISTER'}, to=State{id='REGISTER'}} = 1
63  Transition{from=State{id='REGISTER'}, to=State{id='DASHBOARD'}} = 32
    Transition{from=State{id='REGISTER'}, to=State{id='LOGIN'}} = 2
65  Transition{from=State{id='REGISTER'}, to=State{id='OUTSIDE'}} = 10
    Total time of users spent in this state: 825sec
67  ---------------------------------------------------------------------
    Second order Markov model transitions:
69  SecondOrderTransiton{from=State{id='START'}, current=State{id='REGISTER'}, next=State{id='DASHBOARD
        '}, transition=23}
    SecondOrderTransiton{from=State{id='START'}, current=State{id='REGISTER'}, next=State{id='LOGIN'},
        transition=1}
71  SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='REGISTER'}, next=State{id='
        REGISTER'}, transition=1}
    SecondOrderTransiton{from=State{id='REGISTER'}, current=State{id='REGISTER'}, next=State{id='
        DASHBOARD'}, transition=1}
73  SecondOrderTransiton{from=State{id='ABOUT'}, current=State{id='REGISTER'}, next=State{id='DASHBOARD
        '}, transition=2}
    SecondOrderTransiton{from=State{id=''}, current=State{id='REGISTER'}, next=State{id='LOGIN'},
        transition=1}
75  SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='REGISTER'}, next=State{id='DASHBOARD
        '}, transition=6}
    ---------------------------------------------------------------------
77  State: State{id='LIKE'}
    Think-time in ms (normal distributed):
79  Mean: 202.6066350710902
    StandardDeviation: 655.6210690963171
81  Transition{from=State{id='LIKE'}, to=State{id='PROFILE'}} = 1
    Transition{from=State{id='LIKE'}, to=State{id='LIKE'}} = 15
83  Transition{from=State{id='LIKE'}, to=State{id='LIKED'}} = 827
    Transition{from=State{id='LIKE'}, to=State{id='DASHBOARD'}} = 1
85  Total time of users spent in this state: 171sec
    ---------------------------------------------------------------------
87  Second order Markov model transitions:
    SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='LIKE'}, next=State{id='LIKED'},
        transition=793}
89  SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='LIKE'}, next=State{id='PROFILE'},
        transition=1}
    SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='LIKE'}, next=State{id='LIKED'},
        transition=7}
91  SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='LIKE'}, next=State{id='LIKED'},
        transition=8}
    SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='LIKE'}, next=State{id='LIKE'},
        transition=10}
93  SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='LIKE'}, next=State{id='LIKED'},
        transition=13}
    SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='LIKE'}, next=State{id='LIKE'},
        transition=3}
95  SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='LIKE'}, next=State{id='LIKED'},
        transition=6}
    SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='LIKE'}, next=State{id='DASHBOARD'},
         transition=1}
97  SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='LIKE'}, next=State{id='LIKE'},
        transition=2}
    ---------------------------------------------------------------------
99  State: State{id='DASHBOARD'}
    Think-time in ms (normal distributed):
101 Mean: 4908.713692946059
    StandardDeviation: 7626.955071600951
103 Transition{from=State{id='DASHBOARD'}, to=State{id='NEARBY'}} = 166
    Transition{from=State{id='DASHBOARD'}, to=State{id='PROFILE'}} = 104
105 Transition{from=State{id='DASHBOARD'}, to=State{id='LOGIN'}} = 1
    Transition{from=State{id='DASHBOARD'}, to=State{id='PUBLISH'}} = 160
107 Transition{from=State{id='DASHBOARD'}, to=State{id='LOGOUT'}} = 50
    Transition{from=State{id='DASHBOARD'}, to=State{id='START'}} = 1
109 Transition{from=State{id='DASHBOARD'}, to=State{id='OUTSIDE'}} = 42
    Total time of users spent in this state: 2366sec
111 ---------------------------------------------------------------------
    Second order Markov model transitions:
113 SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='DASHBOARD'}, next=State{id='PUBLISH'
        }, transition=14}
```

```
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='DASHBOARD'}, next=State{id='
    PUBLISH'}, transition=27}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='DASHBOARD'}, next=State{id='
    NEARBY'}, transition=9}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='DASHBOARD'}, next=State{id='NEARBY'
    }, transition=16}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='DASHBOARD'}, next=State{id='PUBLISH'
    }, transition=10}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='DASHBOARD'}, next=State{id='
    PUBLISH'}, transition=76}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='DASHBOARD'}, next=State{id='LOGIN'
    }, transition=1}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='DASHBOARD'}, next=State{id='PROFILE'
    }, transition=22}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='DASHBOARD'}, next=State{id='NEARBY
    '}, transition=27}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='DASHBOARD'}, next=State{id='PROFILE
    '}, transition=55}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='DASHBOARD'}, next=State{id='NEARBY'
    }, transition=54}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='DASHBOARD'}, next=State{id='NEARBY'
    }, transition=29}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='DASHBOARD'}, next=State{id='LOGOUT'
    }, transition=33}
SecondOrderTransiton{from=State{id='REGISTER'}, current=State{id='DASHBOARD'}, next=State{id='
    NEARBY'}, transition=16}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='DASHBOARD'}, next=State{id='OUTSIDE'
    }, transition=2}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='DASHBOARD'}, next=State{id='NEARBY
    '}, transition=13}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='DASHBOARD'}, next=State{id='LOGOUT
    '}, transition=12}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='DASHBOARD'}, next=State{id='
    OUTSIDE'}, transition=9}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='DASHBOARD'}, next=State{id='PROFILE'
    }, transition=7}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='DASHBOARD'}, next=State{id='OUTSIDE
    '}, transition=10}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='DASHBOARD'}, next=State{id='PUBLISH
    '}, transition=17}
SecondOrderTransiton{from=State{id='REGISTER'}, current=State{id='DASHBOARD'}, next=State{id='
    PUBLISH'}, transition=11}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='DASHBOARD'}, next=State{id='START'
    }, transition=1}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='DASHBOARD'}, next=State{id='
    PROFILE'}, transition=13}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='DASHBOARD'}, next=State{id='
    OUTSIDE'}, transition=2}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='DASHBOARD'}, next=State{id='
    PUBLISH'}, transition=5}
SecondOrderTransiton{from=State{id='REGISTER'}, current=State{id='DASHBOARD'}, next=State{id='
    PROFILE'}, transition=4}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='DASHBOARD'}, next=State{id='NEARBY
    '}, transition=2}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='DASHBOARD'}, next=State{id='
    PROFILE'}, transition=1}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='DASHBOARD'}, next=State{id='LOGOUT'
    }, transition=1}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='DASHBOARD'}, next=State{id='
    PROFILE'}, transition=1}
SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='DASHBOARD'}, next=State{id='PROFILE'
    }, transition=1}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='DASHBOARD'}, next=State{id='LOGOUT'
    }, transition=3}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='DASHBOARD'}, next=State{id='LOGOUT
    '}, transition=1}
SecondOrderTransiton{from=State{id='REGISTER'}, current=State{id='DASHBOARD'}, next=State{id='
    OUTSIDE'}, transition=1}
------------------------------------------------------------------------
State: State{id='PUBLISHED'}
Think-time in ms (normal distributed):
Mean: 11733.812949640285
StandardDeviation: 18667.40767404802
```

```
153 Transition{from=State{id='PUBLISHED'}, to=State{id='PUBLISH'}} = 3
    Transition{from=State{id='PUBLISHED'}, to=State{id='OUTSIDE'}} = 16
155 Transition{from=State{id='PUBLISHED'}, to=State{id='DASHBOARD'}} = 86
    Transition{from=State{id='PUBLISHED'}, to=State{id='NEARBY'}} = 21
157 Transition{from=State{id='PUBLISHED'}, to=State{id='PROFILE'}} = 29
    Total time of users spent in this state: 1631sec
159 ---------------------------------------------------------------------
    Second order Markov model transitions:
161 SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISHED'}, next=State{id='
        PROFILE'}, transition=25}
    SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISHED'}, next=State{id='
        DASHBOARD'}, transition=81}
163 SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PUBLISHED'}, next=State{id='
        DASHBOARD'}, transition=1}
    SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISHED'}, next=State{id='
        PUBLISH'}, transition=3}
165 SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISHED'}, next=State{id='NEARBY
        '}, transition=20}
    SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PUBLISHED'}, next=State{id='
        PROFILE'}, transition=4}
167 SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISHED'}, next=State{id='
        OUTSIDE'}, transition=9}
    SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PUBLISHED'}, next=State{id='
        OUTSIDE'}, transition=1}
169 SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PUBLISHED'}, next=State{id='
        DASHBOARD'}, transition=4}
    SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PUBLISHED'}, next=State{id='NEARBY
        '}, transition=1}
171 ---------------------------------------------------------------------
    State: State{id='PROFILE'}
173 Think-time in ms (normal distributed):
    Mean: 16792.134831460673
175 StandardDeviation: 19544.1422710658
    Transition{from=State{id='PROFILE'}, to=State{id='LOGIN'}} = 2
177 Transition{from=State{id='PROFILE'}, to=State{id='LOGOUT'}} = 1
    Transition{from=State{id='PROFILE'}, to=State{id='DASHBOARD'}} = 95
179 Transition{from=State{id='PROFILE'}, to=State{id='NEARBY'}} = 66
    Transition{from=State{id='PROFILE'}, to=State{id='PUBLISH'}} = 4
181 Transition{from=State{id='PROFILE'}, to=State{id='PROFILE'}} = 3
    Transition{from=State{id='PROFILE'}, to=State{id='OUTSIDE'}} = 27
183 Transition{from=State{id='PROFILE'}, to=State{id='LIKE'}} = 7
    Total time of users spent in this state: 2989sec
185 ---------------------------------------------------------------------
    Second order Markov model transitions:
187 SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='PROFILE'}, next=State{id='
        DASHBOARD'}, transition=13}
    SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PROFILE'}, next=State{id='NEARBY'},
         transition=21}
189 SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='
        DASHBOARD'}, transition=51}
    SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='
        OUTSIDE'}, transition=13}
191 SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='NEARBY
        '}, transition=24}
    SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PROFILE'}, next=State{id='NEARBY'
        }, transition=1}
193 SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='PROFILE'}, next=State{id='NEARBY
        '}, transition=16}
    SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PROFILE'}, next=State{id='PUBLISH'
        }, transition=2}
195 SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PROFILE'}, next=State{id='
        DASHBOARD'}, transition=4}
    SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PROFILE'}, next=State{id='NEARBY'
        }, transition=3}
197 SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PROFILE'}, next=State{id='DASHBOARD
        '}, transition=23}
    SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='PROFILE'}, next=State{id='LIKE'},
        transition=1}
199 SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='
        PUBLISH'}, transition=2}
    SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PROFILE'}, next=State{id='OUTSIDE'
        }, transition=2}
```

```
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PROFILE'}, next=State{id='DASHBOARD'}, transition=2}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='PROFILE'}, next=State{id='PROFILE'}, transition=1}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='PROFILE'}, next=State{id='DASHBOARD'}, transition=2}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='PROFILE'}, transition=2}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PROFILE'}, next=State{id='LIKE'}, transition=5}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='LOGOUT'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='LOGIN'}, transition=2}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='PROFILE'}, next=State{id='NEARBY'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PROFILE'}, next=State{id='LIKE'}, transition=1}
------------------------------------------------------------------------
State: State{id='PUBLISH'}
Think-time in ms (normal distributed):
Mean: 36875.0
StandardDeviation: 28804.29035503466
Transition{from=State{id='PUBLISH'}, to=State{id='OUTSIDE'}} = 18
Transition{from=State{id='PUBLISH'}, to=State{id='PUBLISHED'}} = 144
Transition{from=State{id='PUBLISH'}, to=State{id=''}} = 1
Transition{from=State{id='PUBLISH'}, to=State{id='LOGOUT'}} = 1
Transition{from=State{id='PUBLISH'}, to=State{id='PROFILE'}} = 6
Transition{from=State{id='PUBLISH'}, to=State{id='NEARBY'}} = 11
Transition{from=State{id='PUBLISH'}, to=State{id='DASHBOARD'}} = 2
Transition{from=State{id='PUBLISH'}, to=State{id='PUBLISH'}} = 11
Total time of users spent in this state: 6490sec
------------------------------------------------------------------------
Second order Markov model transitions:
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id='PUBLISHED'}, transition=125}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id='NEARBY'}, transition=9}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id='PROFILE'}, transition=5}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PUBLISH'}, next=State{id='PUBLISH'}, transition=2}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISH'}, next=State{id='PUBLISHED'}, transition=7}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='PUBLISH'}, next=State{id='PROFILE'}, transition=1}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PUBLISH'}, next=State{id='PUBLISHED'}, transition=4}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='PUBLISH'}, next=State{id='PUBLISHED'}, transition=1}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='PUBLISH'}, next=State{id='PUBLISHED'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id='OUTSIDE'}, transition=11}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='PUBLISH'}, next=State{id='OUTSIDE'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id='LOGOUT'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id='PUBLISH'}, transition=5}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id='DASHBOARD'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PUBLISH'}, next=State{id='PUBLISHED'}, transition=6}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='PUBLISH'}, next=State{id='PUBLISH'}, transition=1}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISH'}, next=State{id='PUBLISH'}, transition=2}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='PUBLISH'}, next=State{id='NEARBY'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='PUBLISH'}, next=State{id=''}, transition=1}
```

```
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PUBLISH'}, next=State{id='DASHBOARD'}, transition=1}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='PUBLISH'}, next=State{id='OUTSIDE'}, transition=2}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='PUBLISH'}, next=State{id='PUBLISH'}, transition=1}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='PUBLISH'}, next=State{id='NEARBY'}, transition=1}
----------------------------------------------------------------------
State: State{id='LOGIN'}
Think-time in ms (normal distributed):
Mean: 10847.619047619046
StandardDeviation: 10174.233057828895
Transition{from=State{id='LOGIN'}, to=State{id='DASHBOARD'}} = 93
Transition{from=State{id='LOGIN'}, to=State{id='START'}} = 2
Transition{from=State{id='LOGIN'}, to=State{id='ABOUT'}} = 1
Transition{from=State{id='LOGIN'}, to=State{id='REGISTER'}} = 6
Transition{from=State{id='LOGIN'}, to=State{id='OUTSIDE'}} = 10
Transition{from=State{id='LOGIN'}, to=State{id='LOGIN'}} = 3
Total time of users spent in this state: 1139sec
----------------------------------------------------------------------
Second order Markov model transitions:
SecondOrderTransiton{from=State{id='START'}, current=State{id='LOGIN'}, next=State{id='DASHBOARD'}, transition=60}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='LOGIN'}, next=State{id='DASHBOARD'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='LOGIN'}, next=State{id='DASHBOARD'}, transition=23}
SecondOrderTransiton{from=State{id='REGISTER'}, current=State{id='LOGIN'}, next=State{id='DASHBOARD'}, transition=2}
SecondOrderTransiton{from=State{id='ABOUT'}, current=State{id='LOGIN'}, next=State{id='DASHBOARD'}, transition=2}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='LOGIN'}, next=State{id='START'}, transition=1}
SecondOrderTransiton{from=State{id='START'}, current=State{id='LOGIN'}, next=State{id='ABOUT'}, transition=1}
SecondOrderTransiton{from=State{id=''}, current=State{id='LOGIN'}, next=State{id='DASHBOARD'}, transition=4}
SecondOrderTransiton{from=State{id='START'}, current=State{id='LOGIN'}, next=State{id='REGISTER'}, transition=4}
SecondOrderTransiton{from=State{id='START'}, current=State{id='LOGIN'}, next=State{id='LOGIN'}, transition=2}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='LOGIN'}, next=State{id='REGISTER'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='LOGIN'}, next=State{id='REGISTER'}, transition=1}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='LOGIN'}, next=State{id='DASHBOARD'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='LOGIN'}, next=State{id='LOGIN'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='LOGIN'}, next=State{id='OUTSIDE'}, transition=1}
SecondOrderTransiton{from=State{id='START'}, current=State{id='LOGIN'}, next=State{id='START'}, transition=1}
----------------------------------------------------------------------
State: State{id='NEARBY'}
Think-time in ms (normal distributed):
Mean: 8334.291187739462
StandardDeviation: 6973.469266458318
Transition{from=State{id='NEARBY'}, to=State{id='LIKED'}} = 1
Transition{from=State{id='NEARBY'}, to=State{id='NEARBY'}} = 21
Transition{from=State{id='NEARBY'}, to=State{id='START'}} = 1
Transition{from=State{id='NEARBY'}, to=State{id='DASHBOARD'}} = 155
Transition{from=State{id='NEARBY'}, to=State{id='OUTSIDE'}} = 53
Transition{from=State{id='NEARBY'}, to=State{id='LIKE'}} = 805
Transition{from=State{id='NEARBY'}, to=State{id='PUBLISH'}} = 8
Transition{from=State{id='NEARBY'}, to=State{id='PUBLISHED'}} = 1
Transition{from=State{id='NEARBY'}, to=State{id='PROFILE'}} = 51
Transition{from=State{id='NEARBY'}, to=State{id=''}} = 1
Total time of users spent in this state: 8701sec
----------------------------------------------------------------------
Second order Markov model transitions:
```

```
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=103}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='NEARBY'}, next=State{id='NEARBY'}, transition=4}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='NEARBY'}, next=State{id='PROFILE'}, transition=1}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=48}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='NEARBY'}, next=State{id='PUBLISHED'}, transition=1}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=94}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=34}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='NEARBY'}, next=State{id='OUTSIDE'}, transition=1}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=617}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='NEARBY'}, next=State{id='OUTSIDE'}, transition=14}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='NEARBY'}, next=State{id='PUBLISH'}, transition=5}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='NEARBY'}, next=State{id='PROFILE'}, transition=14}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=4}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=16}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=3}
SecondOrderTransiton{from=State{id='START'}, current=State{id='NEARBY'}, next=State{id='OUTSIDE'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=13}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='NEARBY'}, next=State{id='PROFILE'}, transition=5}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=9}
SecondOrderTransiton{from=State{id='PUBLISHED'}, current=State{id='NEARBY'}, next=State{id='OUTSIDE'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=6}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='NEARBY'}, next=State{id='START'}, transition=1}
SecondOrderTransiton{from=State{id=''}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=1}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='NEARBY'}, next=State{id='PROFILE'}, transition=3}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='NEARBY'}, next=State{id='PROFILE'}, transition=27}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='NEARBY'}, next=State{id='OUTSIDE'}, transition=2}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='NEARBY'}, next=State{id='OUTSIDE'}, transition=3}
SecondOrderTransiton{from=State{id='START'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='NEARBY'}, next=State{id='LIKED'}, transition=1}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='NEARBY'}, next=State{id='PROFILE'}, transition=1}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=5}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='NEARBY'}, next=State{id='NEARBY'}, transition=2}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='NEARBY'}, next=State{id='NEARBY'}, transition=7}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='NEARBY'}, next=State{id=''}, transition=1}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=3}
SecondOrderTransiton{from=State{id='LIKED'}, current=State{id='NEARBY'}, next=State{id='NEARBY'}, transition=4}
```

```
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='NEARBY'}, next=State{id='PUBLISH'}, transition=1}
SecondOrderTransiton{from=State{id='START'}, current=State{id='NEARBY'}, next=State{id='DASHBOARD'}, transition=1}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='NEARBY'}, next=State{id='LIKE'}, transition=2}
SecondOrderTransiton{from=State{id='PROFILE'}, current=State{id='NEARBY'}, next=State{id='NEARBY'}, transition=2}
SecondOrderTransiton{from=State{id='PUBLISH'}, current=State{id='NEARBY'}, next=State{id='NEARBY'}, transition=2}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='NEARBY'}, next=State{id='PUBLISH'}, transition=2}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='NEARBY'}, next=State{id='OUTSIDE'}, transition=1}
------------------------------------------------------------------------
State: State{id='START'}
Think-time in ms (normal distributed):
Mean: 9779.661016949152
StandardDeviation: 14280.544169059185
Transition{from=State{id='START'}, to=State{id='REGISTER'}} = 30
Transition{from=State{id='START'}, to=State{id='START'}} = 2
Transition{from=State{id='START'}, to=State{id='LOGIN'}} = 71
Transition{from=State{id='START'}, to=State{id='NEARBY'}} = 3
Transition{from=State{id='START'}, to=State{id='ABOUT'}} = 12
Transition{from=State{id='START'}, to=State{id='OUTSIDE'}} = 42
Total time of users spent in this state: 1154sec
------------------------------------------------------------------------
Second order Markov model transitions:
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='START'}, next=State{id='LOGIN'}, transition=64}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='START'}, next=State{id='REGISTER'}, transition=27}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='START'}, next=State{id='ABOUT'}, transition=9}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='START'}, next=State{id='NEARBY'}, transition=3}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='START'}, next=State{id='OUTSIDE'}, transition=9}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='START'}, next=State{id='LOGIN'}, transition=1}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='START'}, next=State{id='LOGIN'}, transition=1}
SecondOrderTransiton{from=State{id='DASHBOARD'}, current=State{id='START'}, next=State{id='ABOUT'}, transition=1}
SecondOrderTransiton{from=State{id='ABOUT'}, current=State{id='START'}, next=State{id='LOGIN'}, transition=4}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='START'}, next=State{id='START'}, transition=2}
SecondOrderTransiton{from=State{id='START'}, current=State{id='START'}, next=State{id='OUTSIDE'}, transition=1}
SecondOrderTransiton{from=State{id='LOGOUT'}, current=State{id='START'}, next=State{id='REGISTER'}, transition=1}
SecondOrderTransiton{from=State{id='LOGOUT'}, current=State{id='START'}, next=State{id='LOGIN'}, transition=1}
SecondOrderTransiton{from=State{id='START'}, current=State{id='START'}, next=State{id='REGISTER'}, transition=1}
SecondOrderTransiton{from=State{id='LOGOUT'}, current=State{id='START'}, next=State{id='ABOUT'}, transition=2}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='START'}, next=State{id='REGISTER'}, transition=1}
------------------------------------------------------------------------
State: State{id='OUTSIDE'}
Think-time in ms (normal distributed):
Mean: NaN
StandardDeviation: NaN
Transition{from=State{id='OUTSIDE'}, to=State{id=''}} = 4
Transition{from=State{id='OUTSIDE'}, to=State{id='PROFILE'}} = 9
Transition{from=State{id='OUTSIDE'}, to=State{id='LIKE'}} = 8
Transition{from=State{id='OUTSIDE'}, to=State{id='REGISTER'}} = 4
Transition{from=State{id='OUTSIDE'}, to=State{id='START'}} = 145
Transition{from=State{id='OUTSIDE'}, to=State{id='ABOUT'}} = 4
Transition{from=State{id='OUTSIDE'}, to=State{id='DASHBOARD'}} = 24
Transition{from=State{id='OUTSIDE'}, to=State{id='LOGOUT'}} = 1
```

```
Transition{from=State{id='OUTSIDE'}, to=State{id='NEARBY'}} = 30
Transition{from=State{id='OUTSIDE'}, to=State{id='PUBLISHED'}} = 10
Transition{from=State{id='OUTSIDE'}, to=State{id='LOGIN'}} = 30
Transition{from=State{id='OUTSIDE'}, to=State{id='PUBLISH'}} = 8
Total time of users spent in this state: 0sec
------------------------------------------------------------------------
Second order Markov model transitions:
------------------------------------------------------------------------
State: State{id='LIKED'}
Think-time in ms (normal distributed):
Mean: 3627.118644067796
StandardDeviation: 1217.6243847222202
Transition{from=State{id='LIKED'}, to=State{id='REGISTER'}} = 1
Transition{from=State{id='LIKED'}, to=State{id='LIKE'}} = 9
Transition{from=State{id='LIKED'}, to=State{id='NEARBY'}} = 778
Transition{from=State{id='LIKED'}, to=State{id='PROFILE'}} = 2
Transition{from=State{id='LIKED'}, to=State{id='DASHBOARD'}} = 36
Transition{from=State{id='LIKED'}, to=State{id='OUTSIDE'}} = 2
Total time of users spent in this state: 2996sec
------------------------------------------------------------------------
Second order Markov model transitions:
SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='LIKED'}, next=State{id='DASHBOARD'},
     transition=35}
SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='LIKED'}, next=State{id='NEARBY'},
     transition=778}
SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='LIKED'}, next=State{id='REGISTER'},
     transition=1}
SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='LIKED'}, next=State{id='LIKE'},
     transition=9}
SecondOrderTransiton{from=State{id='LIKE'}, current=State{id='LIKED'}, next=State{id='PROFILE'},
     transition=2}
SecondOrderTransiton{from=State{id='NEARBY'}, current=State{id='LIKED'}, next=State{id='DASHBOARD'
     }, transition=1}
------------------------------------------------------------------------
State: State{id='ABOUT'}
Think-time in ms (normal distributed):
Mean: 23625.0
StandardDeviation: 19798.538907130063
Transition{from=State{id='ABOUT'}, to=State{id='START'}} = 4
Transition{from=State{id='ABOUT'}, to=State{id='OUTSIDE'}} = 9
Transition{from=State{id='ABOUT'}, to=State{id='LOGIN'}} = 2
Transition{from=State{id='ABOUT'}, to=State{id='REGISTER'}} = 2
Total time of users spent in this state: 189sec
------------------------------------------------------------------------
Second order Markov model transitions:
SecondOrderTransiton{from=State{id='START'}, current=State{id='ABOUT'}, next=State{id='OUTSIDE'},
     transition=3}
SecondOrderTransiton{from=State{id='OUTSIDE'}, current=State{id='ABOUT'}, next=State{id='LOGIN'},
     transition=1}
SecondOrderTransiton{from=State{id='START'}, current=State{id='ABOUT'}, next=State{id='START'},
     transition=4}
SecondOrderTransiton{from=State{id='START'}, current=State{id='ABOUT'}, next=State{id='REGISTER'},
     transition=2}
SecondOrderTransiton{from=State{id='LOGIN'}, current=State{id='ABOUT'}, next=State{id='LOGIN'},
     transition=1}
```

List of Figures

1.1 UML Use-Case-diagram of the Travelistr software system. 6
1.2 Sample UML Sequence-diagram for the Use-Case *Publish picture* of the Travelistr software system (Error cases are left out for simplification). 7
1.3 UML Component-diagram of the Travelistr software system. 8

5.1 The workflow for performance analysis that is used widely in research and is subject of the first part of this literature review (partly based on [86, p. 1530, Fig. 1]). 30
5.2 Simplified class diagram of the CSM metamodel, taken from [86](p. 1536, Fig. 6). 31
5.3 Class diagram of the PMIF2 metamodel, taken from [74] 32
5.4 Class diagram of the KLAPER metamodel, taken from [45]. 33
5.5 Class diagram of the LQN metamodel, taken from [44](p. 3, Fig. 1.2). 34
5.6 Class diagram of the UML SPT performance analysis domain model, taken from [62]. 35

6.1 The model-measure-feedback cycle is the basic workflow of the ASPE-approach. 44
6.2 The CETO metamodel. 48
6.3 The MUPOM metamodel. 49
6.4 The ASPE-workflow that combines the model-based approach and the measurement-based approach. 51
6.5 Hidden Markov Models emissions and states. 54
6.6 Use-case requests and operations on resources resulting in two layered observations. 57

7.1 Markov model representing the states of Travelistr and the expected user-behaviour defined with predictive knowledge in Sprint 0. 64
7.2 *getUser()* - service times in ms measured on the deployed Travelistr-system after Sprint 1, n = 416. 70
7.3 Service times of operation *getUser()* measured on the deployed Travelistr-system after Sprint 1, n = 1001. 71
7.4 Service times of operation *saveUser()* measured on the deployed Travelistr-system after Sprint 1, n = 429. 71

7.5 Response times of the different already implemented pages at the end of Sprint 1. 72
7.6 Response times of the sub-operations of *publishImage()* and the total response times, n = 176. 74
7.7 Expected Utilization of the resources of the Travelistr software system over time at the end of the development Sprints. 77
7.8 Results of the empirical user test in form of a first order Markov model. . . . 79

List of Tables

6.1 Corresponding HMM and software system elements for unveiling Markov models given incomplete information. 55

7.1 Predicted transition matrix of user-behaviour before implementation in Sprint 0. 65
7.2 Results of steady state analysis and derived arrival rates to the Travelistr system given predictive knowledge in Sprint 0. 66
7.3 Distinct set of Travelistr operations and their service times in milliseconds predicted before implementation in Sprint 0. 67
7.4 Resource-utilization: Average number of users in each station DB, CPU and ImageServer in the Travelistr system calculated with predictive knowledge in Sprint 0. 67
7.5 Travelistr operations and their service times in milliseconds observed on the deployed software system at the end of Sprint 1. 73
7.6 Distinct set of measured Travelistr operations of the deployed system after Sprint 2 and their service times in milliseconds. 75
7.7 Distinct set of measured Travelistr operations of the deployed system and their service times in milliseconds after Sprint 3. 77
7.8 Results of comparing the first-order Markov model solution of user-behaviour with the second-order Markov model. 80
7.9 Observed total times summed of each user that were spent in each state of the Travelistr system. 81

Bibliography

[1] 10 companies killing it at DevOps, Teachbeacon, HPE Software initiative. http://techbeacon.com/10-companies-killing-it-devopsl. Accessed: 2016-06-04.

[2] Architecture Tradeoff Analysis Method ATAM. www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm. Accessed: 2016-11-30.

[3] ATL Transformation Language. https://eclipse.org/atl/. Accessed: 2016-11-30.

[4] The chef automation environment. https://www.chef.io/chef/. Accessed: 2016-06-30.

[5] Cloudinary: Image and video management in the cloud. http://cloudinary.com/. Accessed: 2016-11-30.

[6] DigitalOcean: Cloud computing, designed for developers. https://www.digitalocean.com/. Accessed: 2016-11-30.

[7] Disaster Girl, Worked fine in Dev - Ops Problem Now. https://memegenerator.net/instance/22605665. Accessed: 2016-11-30.

[8] Empiric data and case-study findings as well as the Travelistr software system. http://46.101.118.228:8080/TravelistrApp/. Accessed: 2016-11-30.

[9] Gartner Outlook 2018. http://www.gartner.com/newsroom/id/2939217. Accessed: 2016-06-30.

[10] Juju charms. https://jujucharms.com/. Accessed: 2016-06-30.

[11] Kieker tools for software monitoring. http://kieker-monitoring.net/. Accessed: 2016-11-30.

[12] A list of queuing theory software. http://web2.uwindsor.ca/math/hlynka/qsoft.html. Accessed: 2016-05-03.

[13] OMGs Model Driven Architecture. http://www.omg.org/mda/. Accessed: 2016-06-30.

[14] PMIF Schemas. http://www.spe-ed.com/pmif/PMIF/Schemas.html. Accessed: 2016-11-30.

[15] Sigar API: System Information Gatherer And Reporter. https://support.hyperic.com/. Accessed: 2016-11-30.

[16] The Agile Manifesto. http://agilemanifesto.org/principles.html. Accessed: 2016-05-03.

[17] The Apache JMeter application. http://jmeter.apache.org/. Accessed: 2016-11-30.

[18] The Kieker project on Sorceforge. https://sourceforge.net/projects/kieker/. Accessed: 2016-11-30.

[19] The MetaObject Facility Specification. http://www.omg.org/mof/. Accessed: 2016-11-30.

[20] The ModaClouds research project. http://www.modaclouds.eu/. Accessed: 2016-06-30.

[21] The OASIS TOSCA Specification v 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html. Accessed: 2016-06-04.

[22] The OpenTOSCA Ecosystem. http://www.iaas.uni-stuttgart.de/OpenTOSCA/. Accessed: 2016-06-30.

[23] The UML for MARTE Specification. http://www.omg.org/spec/MARTE/1.1/. Accessed: 2016-06-30.

[24] UML SPT Specification. http://www.omg.org/spec/SPTP/1.1/. Accessed: 2016-11-30.

[25] S. Balsamo. Product form queueing networks. In *Performance Evaluation: Origins and Directions*, pages 377–401. Springer, 2000.

[26] L. Barardinelli, E. Mätzler, T. Mayerhofer, and M. Wimmer. Integrating Performance Modeling in Industrial Automation through AutomationML and PMIF. In *Proceedings of the International Conference on Industrial Informatics (INDIN 2016)*, pages 1–6. IEEE, 2016.

[27] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI (Operating Systems Design and Implementation)*, volume 4, pages 18–18, 2004.

[28] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[29] S. Bernardi, J. Merseguer, and D. C. Petriu. *Model-driven dependability assessment of software systems*. Springer, 2013.

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

[31] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. TOSCA: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

[32] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[33] J. Borges and M. Levene. Data mining of user navigation patterns. In *Web usage analysis and user profiling*, pages 92–112. Springer, 2000.

[34] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice*, volume 1. Morgan & Claypool Publishers, 2012.

[35] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[36] A. Brogi, J. Soldani, and P. Wang. TOSCA in a nutshell: Promises and perspectives. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8745 LNCS:171–186, 2014.

[37] V. Cortellessa, S. Di Gregorio, and A. Di Marco. Using ATL for Transformations in Software Performance Engineering: A Step Ahead of Java-based Transformations? In *Proceedings of the 7th International Workshop on Software and Performance*, WOSP '08, pages 127–132, New York, 2008. ACM.

[38] A. Di Marco and P. Inverardi. Compositional generation of software architecture performance QN models. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 37–46, 2004.

[39] B. Di Martino, G. Cretella, and A. Esposito. Cloud Portability and Interoperability. In *Cloud Portability and Interoperability*, pages 1–14. Springer, 2015.

[40] A. Dyck, R. Penners, and H. Lichter. Towards Definitions for Release Engineering and DevOps. *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 3–3, 2015.

[41] E. d. S. e Silva, R. M. M. Leão, and R. R. Muntz. Performance evaluation with hidden markov models. In *Performance Evaluation of Computer and Communication Systems. Milestones and Future Challenges*, pages 112–128. Springer, 2011.

[42] S. R. Eddy. What is a hidden Markov model? *NATURE BIOTECHNOLOGY*, 22, Number 10, 2004.

[43] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35:148–161, 2009.

[44] G. Franks, P. Maly, M. Woodside, D. Petriu, A. Hubbard, and M. Mroz. Layered Queueing Network Solver and Simulator User Manual. `http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf`, 2013. Accessed: 2016-11-30.

[45] V. Grassi, R. Mirandola, and A. Sabetta. From Design to Analysis Models : a Kernel Language for Performance and Reliability Analysis of Component-based Systems. *Proceedings of the 5th International Workshop on Software and Performance (WOSP 2005)*, pages 25–36, 2005.

[46] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, New York, NY, USA, 1st edition, 2013.

[47] G. Hevizi, M. Biczó, B. Poczos, Z. Szabo, B. Takics, and A. Lorincz. Hidden Markov model finds behavioral patterns of users working with a headmouse driven writing tool. In *Proceedings of the IEEE International Joint Conference 2004 on Neural Networks*, volume 1. IEEE, 2004.

[48] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[49] M. Hüttermann. Introducing DevOps. In *DevOps for Developers*, pages 15–31. Springer, 2012.

[50] S. Jespersen, T. B. Pedersen, and J. Thorhauge. Evaluating the markov assumption for web usage mining. In *Proceedings of the 5th ACM international workshop on Web information and data management*, pages 82–89. ACM, 2003.

[51] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, 1996.

[52] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings of the Fourth International Conference on Engineering of Complex Computer Systems, 1998. ICECCS'98.*, pages 68–78. IEEE, 1998.

[53] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating service resource consumption from response time measurements. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, page 48.

ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[54] X. Li, M. Parizeau, and R. Plamondon. Training hidden markov models with multiple observations-a combinatorial method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(4):371–377, 2000.

[55] Z. Li and J. Tian. Testing the suitability of Markov chains as Web usage models. In *Proceedings of the 27th annual Computer Software and Applications Conference, COMPSAC 2003.*, pages 356–361. IEEE, 2003.

[56] J. D. C. Little. A Proof for the Queuing Formula: L = λ W. *Operations Research*, 9(3):383–387, 1961.

[57] P. Mell and T. Grance. The NIST definition of cloud computing. *NIST Special Publication*, 145:7, 2011.

[58] D. A. Menasce, V. A. Almeida, L. W. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[59] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[60] G. A. Moreno and C. U. Smith. Performance analysis of real-time component architectures: An enhanced model interchange approach. *Performance Evaluation*, 67(8):612–633, 2010.

[61] J. F. Perez, W. Wang, and G. Casale. Towards a devops approach for software quality engineering. In *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, pages 5–10. ACM, 2015.

[62] D. B. Petriu and M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from uml designs. *Software & Systems Modeling*, 6(2):163–184, 2007.

[63] R. Qasha, J. Cala, and P. Watson. Towards automated workflow deployment in the Cloud using TOSCA. In *Proceedings of the 8th International Conference on Cloud Computing*, pages 1037–1040. IEEE, 2015.

[64] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[65] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[66] L. Rabiner and B.-H. Juang. An introduction to hidden Markov models. *ASSP Magazine, IEEE*, 3(January):4–16, 1986.

[67] J. Roche. Adopting DevOps practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[68] M. Rohr, A. Van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: continuous monitoring and on demand visualization of java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering, SE 2008*, pages 80–85, 2008.

[69] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[70] C. H. Sauer. Approximate solution of queueing networks with simultaneous resource possession. *IBM Journal of Research and Development*, 25(6):894–903, 1981.

[71] C. H. Sauer, E. A. MacNair, and S. Salza. A language for extended queueing network models. *IBM Journal of Research and Development*, 24(6):747–755, 1980.

[72] R. Serfozo. *Basics of applied stochastic processes*. Springer Science & Business Media, 2009.

[73] C. Smith and L. Williams. A performance model interchange format. *Journal of Systems and Software*, 49(1):63–80, 1999.

[74] C. U. Smith and C. M. Lladó. Performance Model Interchange Format (PMIF 2.0): XML definition and implementation. In *Proceedings of the First International Conference on the Quantitative Evaluation of Systems, QEST 2004*, pages 38–47, 2004.

[75] C. U. Smith, C. M. Lladó, V. Cortellessa, A. D. Marco, and L. G. Williams. From UML models to software performance results: an SPE process based on XML interchange formats. In *Proceedings of the 5th international workshop on Software and performance*, pages 87–98, 2005.

[76] C. U. Smith, C. M. Lladó, and R. Puigjaner. Performance model interchange format (pmif 2): A comprehensive approach to queueing network model interoperability. *Performance Evaluation*, 67(7):548–568, 2010.

[77] I. Sommerville. *Software Engineering*. Pearson; 9 edition (March 13, 2010), 2010.

[78] M. Stamp. A revealing introduction to hidden Markov models. *Department of Computer Science San Jose State University*, 2004.

[79] A. Van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In *Proceedings of the SPEC International Performance Evaluation Workshop*, pages 124–143. Springer, 2008.

[80] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering - ICPE '12*, number July, pages 247–248, 2012.

[81] R. H. Von Alan, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[82] W. Wang, J. F. Pérez, and G. Casale. Filling the gap: a tool to automate parameter estimation for software performance models. In *Proceedings of the 1st International Workshop on Quality-Aware DevOps*, pages 31–32. ACM, 2015.

[83] J. Wettinger, U. Breitenbucher, and F. Leymann. Standards-based DevOps automation and integration using TOSCA. In *Proceedings of the 7th International Conference on Utility and Cloud Computing, UCC 2014, 2014 IEEE/ACM*, pages 59–68, 2014.

[84] L. G. Williams and C. U. Smith. Performance evaluation of software architectures. In *Proceedings of the 1st international workshop on Software and performance*, pages 164–177. ACM, 1998.

[85] M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. *Future of Software Engineering (FOSE '07)*, pages 171–187, 2007.

[86] M. Woodside, D. C. Petriu, J. Merseguer, D. B. Petriu, and M. Alhaj. Transformation challenges: from software models to performance models. *Software and Systems Modeling*, 13(4):1529–1552, 2014.

[87] M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (PUMA). *Proceedings of the 5th international workshop on Software and performance WOSP 05*, pp:1–12, 2005.

[88] Q. Zhou, H. Ye, and Z. Ding. Performance Analysis of Web Applications Based on User Navigation. *Physics Procedia*, 24:1319–1328, 2012.