PADERBORN UNIVERSITY
The University for the Information Society

Faculty of Electrical Engineering, Computer Science, and Mathematics

Engineering Self-Adaptive Systems with Simulation-Based Performance Prediction

Matthias Wilhelm Becker

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

June, 2017

Danksagung

Diese Dissertation fasst die wissenschaftlichen Erkenntnisse meiner Promotionszeit zusammen. In meiner dieser Zeit haben mich viele Menschen auf viele unterschiedliche Arten dabei unterstützt, zu ebendiesen Erkenntnissen zu kommen. All diesen Menschen möchte ich für ihre Unterstützung „Danke“ sagen.

Ich möchte mich zuallererst bei meinem Doktorvater Steffen Becker bedanken. Seine Beratung und Kritik haben mir dabei geholfen, meine Dissertation erfolgreich abzuschließen. Ich habe durch die Zusammenarbeit mit ihm sehr viel gelernt.

Ebenso wie Steffen Becker danke ich außerdem Gregor Engels für das Anfertigen eines Gutachtens für diese Dissertation und den weiteren Mitgliedern meiner Prüfungskommission Eric Bodden, Eyke Hüllermeier und Marie Christin Platenius dafür, dass sie sich mit meiner Dissertation auseinandergesetzt haben.

Auch meinen Kollegen aus der Fachgruppe Softwaretechnik von Wilhelm Schäfer und Eric Bodden, der Abteilung Softwaretechnik des Fraunhofer IEM und der Lehrstühle von Gregor Engels und Heike Wehrheim möchte ich für die zahlreichen Diskussionen, das Feedback zu wissenschaftlichen Veröffentlichungen und Vorträgen danken. Insbesondere möchte ich mich bei Sebastian Lehrig, Markus Luckey und Marie Christin Platenius bedanken, mit denen ich nicht nur gemeinsam wissenschaftliche Papiere veröffentlicht habe, sondern mit denen ich meine Dissertation in vielen fachlichen Diskussionen weiterentwickeln konnte. Darüber hinaus haben sich meine Freunde und Kollegen Masud Fazal-Baqaie, Christian Heinzemann, Marie Christin Platenius und Sven Walther die Zeit genommen, um mir viel inhaltliches Feedback zu meiner Dissertation zu geben.

Mein Dank gilt auch den Studierenden, die mit ihren Abschlussarbeiten zu meiner Dissertation beigetragen haben. Besonders möchte ich hier Joachim Meyer danken, der in seiner Masterarbeit die ersten Grundlagen von SimuLizar implementiert hat.

Neben der fachlichen Unterstützung durch meine Kollegen und Studierende, haben mich außerdem Jutta Haupt und Jürgen Maniera während meiner Zeit am Lehrstuhl Softwaretechnik bei den vielen organisatorischen und technischen Aufgaben unterstützt.

Zu guter Letzt möchte ich mich bei meiner Familie, meinen Freunden und meiner Partnerin Veronika bedanken. Meine Eltern Bernadette und Herbert ermutigen mich stets dazu, meine Ziele zu verfolgen und unterstützen mich auf meinem Weg. Ebenso wie meine Eltern ist mir meine Schwester Eva ein großes Vorbild, die mir vorlebt, wie man Erfolg und Familie miteinander in Einklang bringt. Meinen besten Freunden möchte ich insbesondere für die Ablenkungen danken, wenn ich diese benötigt habe, und für das Verständnis, wenn ich keine Zeit für sie hatte. Meiner Partnerin Veronika danke ich dafür, dass sie mit viel Liebe alle Erfolge und Misserfolge in meiner Promotionszeit mit mir geteilt hat. Ohne die fortwährende Ermutigung, die ansteckende Freude und die endlose Liebe meiner Familie, meiner Freunde und meiner Partnerin hätte ich diese Dissertation vermutlich weder begonnen noch beendet.

Abstract

Large information systems nowadays are required to perform on a Web-scale with millions of users. Especially in the business-to-business context, the required performance is even contractually specified in the form of service level objectives (SLOs). Often, only cloud computing platforms, which provide virtually unlimited resources and on-demand resource leasing, make information systems that meet these SLOs possible. However, the pay-per-use leasing models of the cloud computing platforms still enforce a trade-off between the achievement of SLOs on the one hand and an economical operation on the other hand. Self-adaptive systems can solve this trade-off by autonomously adapting the amount of leased resources to the actual demand. Thus, these systems accomplish an economical operation and still achieve the SLOs at the same time.

However, in current practice, self-adaptive systems are developed based on experience of software engineers and rule-of-thumb. Conflicting SLOs or design deficiencies that limit the achievement of the SLOs are often only discovered in late development phases, i. e., in the testing phase or even in the operation phase. Thus, the development of self-adaptive systems is slowed down or is even at risk to fail.

With SimuLizar, we provide a model-driven performance prediction method that supports software engineers in identifying design deficiencies that negatively effect the achievement of service level objectives. For that purpose, we introduce the notion of graded achievement of service level objectives, such that trade-offs between conflicting service level objectives can be revealed and solved. With our method, the achievement of services level objectives becomes predictable early at design-time. Thus, also design deficiencies can be revealed early in self-adaptive system development projects and project failures and delays can be averted.

Zusammenfassung

Von großen Informationssysteme wird zunehmend gefordert, dass diese “Web-scale” sind, also auch für mehrere Millionen Nutzern performant sind. Gerade im Business-to-Business Kontext werden Anforderungen an die Performanz der Informationssysteme vertraglich in Form von Service Level Objectives (SLOs) festgehalten. Oft ermöglichen es nur Cloud Computing Plattformen, welche nahezu unbegrenzt viele Resourcen auf Abruf anbieten, Informationssysteme zu realisieren, die diese SLOs einhalten. Jedoch besteht durch die pay-per-use Kostenmodelle der Plattformen auch ein Zielkonflikt zwischen dem wirtschaftlichen Betrieb der Informationssysteme und dem Erreichen der SLOs. Selbst-adaptive Systeme können diesen Zielkonflikt lösen, indem sie die Menge der gemieteten Ressourcen autonom dem Bedarf anpassen und somit einen wirtschaftlichen Betrieb bei Erreichung der SLOs ermöglichen.

In der gängigen Praxis werden selbst-adaptive Systeme jedoch weitestgehend mithilfe der Erfahrung und Faustregeln von Softwaretechnikern entwickelt. Konflikte zwischen SLOs oder Entwurfsfehler, welche die Erreichung der SLOs verhindern, werden oft nur spät im Entwicklungsprozess entdeckt, z.B. erst in der Testphase oder gar erst im Betrieb. Dadurch kommt es in Projekten zur Entwicklung selbst-adaptiver Systeme zu Verzögerungen oder die Projekte drohen sogar zu scheitern.

Mit SimuLizar stellen wir eine modellgetriebene Methode vor, die es Softwaretechnikern ermöglicht Entwurfsfehler, welche die Erreichung von SLOs negativ beeinflussen, frühzeitig zu erkennen. Zu diesem Zweck definieren wir eine graduelle Erreichung von SLOs, so dass Zielkonflikte zwischen SLOs erkannt und gelöst werden können. Die Erreichung von SLOs lässt sich durch die Methode ebenfalls bereits in der Entwurfsphase vorhersagen. Dadurch werden Verzögerungen und das Scheitern von Projekten zur Entwicklung selbstadaptiver Systeme verhindert.

Contents

1. Introduction **1**

1.1. The Znn.com System 5
1.2. Problem Statement 8
1.3. Scientific Contribution 12
1.4. Overview 15

2. Uncertainty, Imprecision, and Gradedness in Software Engineering **17**

2.1. Definitions and Examples 18
2.1.1. Uncertainty 19
2.1.2. Imprecision 21
2.1.3. Gradedness 22
2.2. Applications in Software Engineering 23
2.2.1. Requirements Engineering 23
2.2.2. Design 25
2.2.3. Implementation 29

3. Software Performance Engineering Foundations **31**

3.1. Model-Driven Software Engineering 32
3.2. Software Metrics and Predictions 35
3.3. Model-Driven Software Performance Engineering 37
3.3.1. Palladio 39

4. Self-Adaptive System Performance Modeling **43**

4.1. Scientific Contributions 45
4.2. Modeling Requirements 46
4.2.1. General Requirements 47
4.2.2. Scope of the Viewpoints 48
4.3. Related Work 54
4.4. Self-Adaptive System Performance Modeling Overview 62
4.4.1. Modeling Viewpoints 63
4.4.2. Modeling Roles 66

4.4.3. Modeling Process . . . 68
4.5. System Type Viewpoint . . . 70
4.5.1. System Architecture Types . . . 70
4.5.2. System Resource Context . . . 74
4.6. Run-Time Viewpoint . . . 77
4.6.1. Initial System Architecture Configuration . . . 77
4.6.2. Initial System Deployment . . . 80
4.6.3. System Usage Context . . . 82
4.7. Self-Adaptation Viewpoint . . . 86
4.7.1. Service Level Objectives . . . 87
4.7.2. Monitor Repository . . . 89
4.7.3. Reconfigurations . . . 93
4.8. Evaluation . . . 99
4.8.1. Questions . . . 101
4.8.2. Setup . . . 103
4.8.3. Results . . . 104
4.8.4. Discussion . . . 104
4.8.5. Threats to Validity . . . 106
4.9. Conclusion . . . 107

5. Scalability and Elasticity Prediction Methods 109
5.1. Scientific Contribution . . . 112
5.2. Prediction Method Requirements . . . 113
5.2.1. General Requirements . . . 114
5.2.2. Requirements for Scalability and Elasticity Prediction . . . 114
5.3. Related Work . . . 115
5.4. Prediction Methods Overview . . . 121
5.5. Self-Adaptive System Formalization . . . 123
5.6. Service Level Objective Formalization . . . 129
5.6.1. Derivation of Service Level Objectives . . . 131
5.6.2. Relaxation of Time . . . 132
5.6.3. Relaxation of Accuracy . . . 136

5.7. Scalability Prediction . . . 145
5.7.1. Formalization . . . 145
5.7.2. Metric Definition . . . 147
5.7.3. Metric Implementation . . . 149
5.7.4. Assumptions and Limitations . . . 151
5.8. Elasticity Prediction . . . 152
5.8.1. Formalization . . . 152
5.8.2. Metric Definitions . . . 154
5.8.3. Metric Implementation . . . 156
5.8.4. Assumptions and Limitations . . . 158
5.9. Evaluation . . . 158
5.9.1. Question . . . 160
5.9.2. Setup . . . 161
5.9.3. Results . . . 162
5.9.4. Discussion . . . 165
5.9.5. Threats to Validity . . . 167
5.10. Conclusion . . . 168

6. Tool Support **171**
6.1. Architecture Overview . . . 173
6.1.1. Main Components . . . 173
6.1.2. External Dependencies . . . 174
6.2. User Interface . . . 176
6.3. Modeling Viewpoints . . . 177
6.3.1. System Type Viewpoint . . . 178
6.3.2. Run-Time Viewpoint . . . 178
6.3.3. Self-Adaptation Viewpoint . . . 179
6.4. Performance Analysis Tool . . . 180
6.4.1. Performance Model Interpreter . . . 182
6.4.2. Performance Simulation . . . 184
6.4.3. Reconfiguration Manager . . . 185
6.5. Scalability and Elasticity Prediction . . . 186
6.5.1. Scalability Prediction . . . 187

6.5.2. Elasticity Prediction . . . 188
6.5.3. Prediction Result Visualization . . . 188

7. Conclusion **191**
7.1. Results and Conclusions . . . 192
7.2. Follow-Up Work . . . 193
7.3. Future Work . . . 195

B. Bibliography **197**
Own Publications . . . 197
Supervised Theses . . . 199
Literature . . . 200
Websites and Standards . . . 216

List of Figures, Tables, and Listings **219**

I. Complete Znn.com Example **227**
I.1. System Element Type View . . . 228
I.1.1. System Architecture Types . . . 228
I.1.2. System Resource Context . . . 229
I.2. Initial Configuration View . . . 229
I.2.1. Initial System Architecture Configuration . . . 229
I.2.2. Initial System Deployment . . . 229
I.2.3. System Usage Context . . . 230
I.3. Reconfiguration View . . . 230
I.3.1. Service Level Objectives . . . 230
I.3.2. Monitor Repository . . . 231
I.3.3. Reconfigurations . . . 231

II. Auxiliary Formalization **235**

"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."

Charles Darwin

1 Introduction

Contents

1.1. The Znn.com System 5

1.2. Problem Statement 8

1.3. Scientific Contribution 12

1.4. Overview 15

Large information systems nowadays are required to perform on a Web-scale [Hai13] with millions of users [Net16, Fac16, Sto16]. A fluent performance with low response times of these systems is an important business success factor [SW03, JYJ13]. Achieving this fluent performance requires a vast amount of computing resources. Resource demands are even increasing in the long term due to a growing user base, resulting in a higher workload to be processed. In the shorter term, however, resource demands are varying due to periodic usage fluctuation, e. g., weekend usage versus mid-week usage, and spontaneous usage bursts. These workload variations are challenging because the usage context in which large information systems need to operate reliably becomes *uncertain* for software engineers.

In the past, large information systems were deployed on expensive, energy-hungry in-house computing centers. More recently, these computing centers have been replaced by cloud computing environments, such as Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) environments [ACC+14]. These cloud computing environments provide virtually unlimited computing resources and also provide technical mechanisms to lease and release computing resources at any time during the operation of a information system. Furthermore, in cloud computing environments, resources are typically offered in a pay-per-use fashion [BAB12], i. e., customers of the cloud computing environments pay for only as many resources as they lease. Thus, high investments for in-house computing centers are not required anymore. In conclusion, cloud computing has the potential to reduce operating costs of information systems, on the one hand, and on the other hand, provide high performance. Still both properties, costs and performance, oppose each other in a trade-off.

The solution of this trade-off depends on the usage context of the information system. In general, it is desirable to lease only the *right* amount of resources to keep costs low while still maintaining a fluent performance. However, to always lease the right amount of resources means that the information system must be able to *autonomously adapt* the amount of resources to the changing usage context. For example, whenever the workload is low the information system shall only lease few resources; when the workload is high, the information system shall lease more resources. The two performance properties *scalability* and *elasticity* characterize a system's quality to autonomously adapt its resource consumption its actual resource demand. Scalability is "the ability of the system to sus-

tain increasing workloads by making use of additional resources." [BLB15] Elasticity is "the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomous manner, such that at each point in time the available resources match the current demand as closely as possible." [HKR13] Scalability can hence be understood as a necessary prerequisite for elasticity according to this definition. Furthermore, the adaptation to workload changes is required to be autonomously triggered by the information system itself, i. e., the system has to be self-adaptive.

Today, developing scalable and elastic information systems that are operated in cloud computing environments is still challenging. On the one hand, information systems are expected to be scalable and elastic. For that purpose, thresholds for individual quality metrics, e. g., maximum operating costs or maximum resource utilization, are usually contractually demanded in service level agreements (SLAs) [TBv04]. The information system is then expected to autonomously lease and release cloud computing resources in order to comply with the SLAs.

On the other hand, those SLAs typically do not reflect the fact that the context of the software system is subject to change at operation time and thus neglect that trade-offs between properties, like costs and performance, can be solved differently well by alternative system designs. Hence, a self-adaptive information system can only be assessed as SLA compliant or not compliant. Conclusions about the quality of the self-adaptation cannot be drawn. Furthermore, no engineering method exists that supports software engineers to assure the scalability and elasticity of information systems right from design-time. Consequently, design flaws in the software architecture are often detected late in the development process or even only at operation. Fixing scalability and elasticity issues at this point is often only possible by revising the software architecture, which results in a delay of the development process and increases the time-to-market and development costs [SHK98].

In multiple research communities, there has been some effort to support software engineers to develop scalable and elastic information systems. In particular, self-adaptation is recognized as an individual engineering concern in most phases of the software life-cycle.

In the requirements elicitation phase, the requirements language RELAX [WSB+10] enables the specification of requirements for self-adaptive systems in uncertain contexts. The formalization of RELAX requirements is based on fuzzy logic [Zad65]; thus taking into account that not all requirements can be fully satisfied at once due to individual trade-offs between requirements depending on the concrete context.

In the design phase, the Adapt Case Modeling Language (ACML) [Luc13] supports software engineers designing high-level architectures of self-adaptive systems. ACML supports functional analysis of self-adaptive behavior for high-level software architectures based on model checking. Non-functional quality properties, however, cannot be analyzed with ACML. Palladio [BKR09] is a model-driven software performance engineering method, which can be used to assess the quality of software systems early in the design phase. In contrast to ACML, however, Palladio is limited to static architectures, i. e., non-adaptive systems.

In the implementation phase of self-adaptive systems, software engineers are supported by Rainbow [GCH+04], for example. Rainbow provides a Java-based framework for implementing component-based self-adaptive systems. Another framework is Descartes [HBK11]. It offers a run-time monitoring and self-adaptation framework that is focused on performance-driven adaptation. Descartes provides a run-time model that reflects the system state during operation and is used for planning self-adaptations of the system itself, i. e., it introduces self-awareness to an information system.

In summary, while several engineering tasks for self-adaptive systems are supported by different methods, the prediction of scalability and elasticity properties of self-adaptive systems early in the design phase has mostly been neglected so far.

The goal of this thesis is to support software engineers designing scalable and elastic information systems by providing a method to identify scalability and elasticity issues early in the design phase. By identifying scalability and elasticity issues already at design-time, expensive redesigns and the resulting project delays can be averted.

In this thesis, we contribute to this goal by proving *SimuLizar*, our model-driven performance engineering method for self-adaptive systems. SimuLizar consists of three parts. The first part is a *performance modeling approach* for self-adaptive systems that

is based on the viewpoints initially presented in [Bec11]. The second part are *model-driven prediction methods* that enable software engineers to assess the scalability and elasticity properties of self-adaptive systems at design-time. This includes a formalization and metrics that enable the evaluation of self-adaptive system architectures. Finally, the third part is *SimuLizar Bench*, a tool that implements the performance modeling approach as well as the scalability and elasticity prediction methods.

The Znn.com system, introduced in the next section, serves as a running example in this thesis to illustrate our modeling approach, and the scalability and elasticity prediction methods within SimuLizar.

1.1. The Znn.com System

The Znn.com system [CGS09] is a specimen system from the self-adaptive systems research community. It represents a typical system in the class of web-based systems, including a typical architecture and typical objectives for this class of systems.

The purpose of the Znn.com system is to provide a news website to its clients. Znn.com presents its news websites either with multimedia content, like pictures and videos, or as text-only websites. Which of either presentation forms will be used depends on the workload the Znn.com system is facing. In any case, the news website shall be served promptly to all clients.

Znn.com is implemented as a client-server system with an N-tier architecture. It is deployed on an infrastructure-as-a-service (IaaS) platform, where computing resources, i. e., virtual machines, can be leased and released at any time during operation. Client requests are distributed to application servers by a *load balancer*.

There are three objectives defined for the Znn.com system: (O1) low response times, (O2) low costs, and (O3) high content fidelity. The news should be served to the clients within a reasonable response time, i. e., the content of a news article should be sent from the Znn.com servers to a client within 3.0 seconds after the client has selected an article. The operation costs of Znn.com should be kept low, e. g., server leasing costs

should be capped to guarantee a profitable operation. Finally, the content fidelity of the news article should be as high as possible to engage clients.

In the following sections, we formulate these objectives as concrete requirements for the Znn.com system and provide a high-level architecture of the system. In Chapter 4, we refine these requirements to service level objectives (SLOs) and we refine the high-level architecture to a more fine-grained self-adaptive system architecture using our performance modeling approach.

Requirements

From the system's objectives O1 to O3, we derive the following set of requirements:

R1 The system shall serve articles requested by clients promptly. That is, the mean response time for a user request shall be less than 3.0 seconds, where the mean response time is calculated in 1 minute batches.

R2 The system's computing resource leasing costs shall be less than USD 5.00 per hour.

R3 The system shall serve news articles with highest possible content fidelity ("multimedia" > "text-only") without violating requirements R1 and R2.

R4 The system shall autonomously lease and release computing resources as required to maintain the requirements R1, R2, and R3.

R5 The system shall autonomously select a content fidelity of the served news articles to maintain the requirements R1, R2, and R3.

Requirements R1 to R3 define quantifiable thresholds for the operation of the Znn.com system. That is, the mean response time is required to be less than 3.0 seconds, the leasing costs shall be less than USD 5.00 per hour, and "multimedia" is the preferred content fidelity. Requirements R4 and R5 define the degree of freedom for the self-adaptation of the Znn.com system within these thresholds. That is, requirements R4 and R5 do not define performance constraints or thresholds but specify which parts of

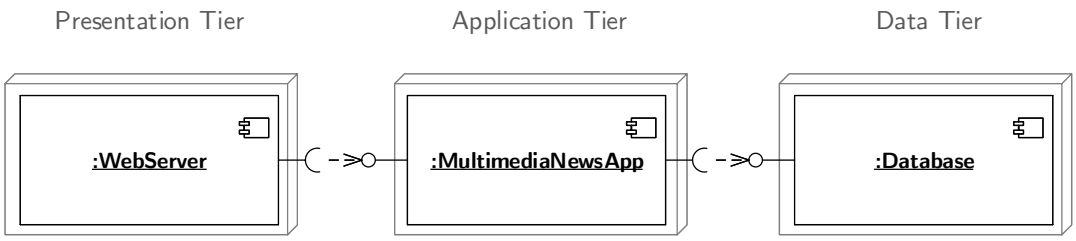


Figure 1.1.: Znn.com components

the system may change, or adapt, in order to be able to satisfy the other requirements, i. e., these requirements relate to the system's scalability and elasticity.

High-Level Architecture

The Znn.com system is implemented as a three tier system, as illustrated in Figure 1.1 with the UML component diagram notation. The system consists of a presentation tier, an application tier, and a data tier.

The presentation tier is deployed on a single virtual machine running a web server. The web server serves hypertext via the HTTP protocol. The hypertext is rendered in a browser on the client device, which is considered outside of the system borders.

The application tier consists of a news application that generates the Znn.com website. The news application is implemented by two alternative, interchangeable components, implementing the same interfaces but generating the website for two different content fidelity levels. The MultimediaNewsApp component generates a rich multimedia website with embedded pictures and videos. The TextualNewsApp generates a text-only website, which requires minimal processing and bandwidth.

Both components, MultimediaNewsApp and TextualNewsApp, are stateless, i. e., the components have no internal state and, if required, a request has to pass its own necessary context. Each instance of these components runs in a dedicated virtual machine, which can be autonomously replicated, started, and stopped during operation.

The data tier consists of a database server component running on a dedicated storage server. In the context of Znn.com, we assume that the database server is not the bottleneck of the system and thus provides enough performance to handle any workload.

The self-adaptive behavior of Znn.com is controlled by rules that describe when virtual machines should be replicated, started, or stopped. Finally, a load balancer (not shown in Figure 1.1) distributes incoming customer requests over all running virtual machines in the application tier.

1.2. Problem Statement

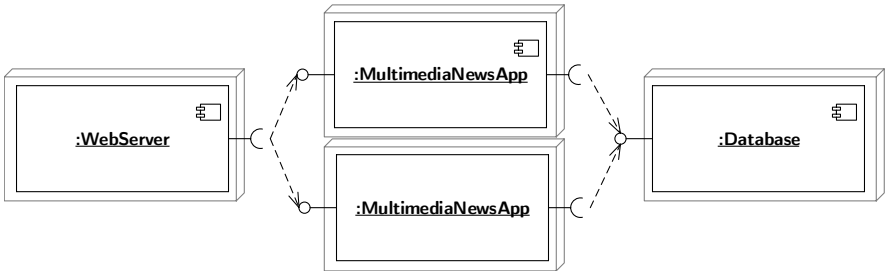
The software architecture of a rather simple system like Znn.com becomes complex due to its self-adaptive behavior. That is, the number of components and their interconnections is variable over time in a self-adaptive system, e. g., due to the leasing and releasing of additional resources and the replication of instances in the application tier.

Each configuration of components and their interconnection represents a different software *architecture configuration*. The transitions between two architecture configurations are called *reconfigurations*. The sum of all architecture reconfigurations is forming a software architecture *reconfiguration space*.

In consequence, the prediction of quality properties like performance for a self-adaptive system becomes challenging: Quality properties have to be predicted for each individual architecture configuration in the reconfiguration space as well as for the transition between architecture configurations during a reconfiguration to assure a constant quality at all times during operation. Existing quality metrics for non-adaptive systems do not capture the quality of the self-adaptive layer in a meaningful way. Quality metrics that capture or try to capture the quality of the self-adaptation layer are often not well defined.

In this section, we outline three particular problems in the engineering process of self-adaptive systems that have not been addressed or solved by other related approaches yet. In general, these problems concern modeling and early performance prediction of self-adaptive systems in the design phase of the software lifecycle.

(a) Architecture configuration with single application tier.



(b) Architecture configuration with replicated application tier.

Figure 1.2.: Alternative system architecture configurations.

Self-Adaptive System Performance Modeling Performance and operating costs of a software system are crucially impacted by the system's architecture. For example, consider two alternative architecture configurations for our Znn.com example system, like illustrated in Figure 1.2.

In the first architecture configuration, illustrated in Figure 1.2a, one instance of the MultimediaNews component is deployed on a single server node. In the second architecture configuration, illustrated in Figure 1.2b, the application tier consists of two instances of the MultimediaNews component. Each instance is deployed on a dedicated server node, which is a replica of the server node in the first architecture configuration.

In scenarios where the processing of requests in the application layer becomes the bottleneck, the second architecture configuration can process more requests than the first architecture configuration can process in the same time. However, the second architecture configuration will also have higher operating costs since two instead of one server nodes have to be operated in the application layer. There is no single architecture configuration that is superior regarding both quality criteria, performance and operating costs, in all scenarios.

Designing multiple or even all individual architecture configurations for each scenario is not feasible either, since the costs to design, implement, and operate individual sys-

tems for each scenario are not justifiable. Instead, it is desired to specify architecture reconfigurations that are triggered and executed autonomously for each scenario, i. e., self-adaptations.

In order to autonomously trigger and execute architecture reconfigurations, the system itself needs (1) self-awareness capabilities to detect situations when a reconfiguration is necessary and (2) the ability to select and execute an appropriate reconfiguration for the detected situation. These two properties can be considered as the *self-adaptation layer*, which enables self-adaptive systems to operate autonomously within a range of scenarios without requiring manual intervention for maintenance. Consequently, these systems can potentially reduce operation costs and also reduce down-times.

The design of self-adaptive system architecture has partly been addressed in research. However, existing modeling approaches, like ACML [Luc13], Palladio [BKR09], and Brun et al. [BDG+09], do not address the specification of performance properties of reconfigurations, such as set up times. Other approaches are focused on run-time models, like Descartes [HBK11], without allowing early analysis of the architecture at design-time. Incerto et al. [ITT15] focus merely on performance properties of self-adaptive systems but do not support designing software system architectures. Further specific aspects of self-adaptive system architectures, like monitoring, have so far only been addressed by few run-time-focused and framework-focused research, such as Descartes [HBK11] and Kieker [vWH12]. A modeling approach that supports all aspects of modeling performance-relevant properties for self-adaptive system architectures is still missing. Consequently, the early assessment of performance properties of self-adaptive system architectures is not possible with current modeling approaches.

Metrics for Scalability and Elasticity To assess and compare the quality of self-adaptive system architectures, specific metrics are required. Software metrics in general provide an objective, reproducible, and quantitative measure to obtain the quality of a software system.

In literature, numerous metrics to obtain the performance of software systems can be found. For example, Bolch [BGTM98] defines response time, throughput, and utilization as standard performance metrics. However, these metrics measure only the quality

of non-adaptive software systems in non-variable operation contexts, i. e., software systems that only have a single software architecture configuration during operation. The performance of the self-adaptation itself, including the transient phase of architecture reconfiguration, cannot be measured with these metrics.

Consider Requirement R1, from Section 1.1, that specifies a performance requirement, i. e., a mean response time of less than 3.0 seconds. A metric, to measure the quality of the performance, is already included in the requirement specification, i. e., the response time.

In contrast, in requirement R4 of the Znn.com example, which specifies a self-adaptive behavior for autonomous resource leasing, a metric is missing. It can be implied that the requirement means that the system shall be scalable and elastic. However, without a concrete metric, the quality of the self-adaptation layer, i. e., the system's autonomous resource leasing quality, cannot be quantified and consequently the quality cannot be assessed objectively. Consequently, software engineers have no means to verify their design decisions or detect design flaws.

In research, there are some definitions of scalability and elasticity metrics that aim to measure the quality of the self-adaptation layer. For example, Herbst et al. define elasticity metrics [HKR13] to measure speed and precision of resource leasing with respect to the resource demand. Folkers et al. [FAS+12] and Islam et al. [ILFL12] provide elasticity metrics in terms of costs. Bondi provides scalability definitions for structural scalability ("ability to expand in a chosen dimension without major modification") and load scalability ("ability of a system to perform gracefully as the offered traffic increases") [Bon00]. However, Bondi does only provide concrete metrics for structural scalability. Further scalability metrics as are defined by Jogalekar and Woodside [JW00] for distributed systems in general. However, these metrics either are defined as benchmark metrics for retrospective evaluation or mix the related metrics scalability, elasticity, and efficiency.

In summary, precisely defined metrics need to be defined in order to quantify scalability and elasticity as dedicated quality properties of self-adaptive systems and to enable early detection of design-flaws in the self-adaptation layer.

Prediction of Scalability and Elasticity The prediction of scalability and elasticity properties at design-time requires a method to obtain the according metrics from a model of the software system. When a software engineer designs a software system, like the Znn.com system, she should apply methods to constructively assure the quality properties, early at design-time. Fixing quality defects of software systems late in the development or even during operation has proven to be cumbersome and expensive [Gla98].

Analytical as well as simulation-based methods to predict performance metrics based on software architectures of non-adaptive systems have already been described and implemented. For example, the Palladio approach [BKR09] provides a model-driven method for prediction of performance metrics like response time, utilization, and waiting times for non-adaptive software architectures. Palladio, however, does not provide a method to predict scalability and elasticity properties yet. In D-KLAPER [GMR09], a model-transformation chain from architecture models to analysis models for self-adaptive systems is described. A concrete method to assess scalability and elasticity metrics from these analysis models is not provided. Another approach by Incerto [ITT15] provides an analytical prediction method for self-adaptive systems, but is limited to the performance metric response time, which is not sufficient to measure the quality of the self-adaptation itself.

Consequently, a model-driven method and tool support is still required to assess scalability and elasticity properties of self-adaptive systems at design-time.

1.3. Scientific Contribution

The contribution of this thesis is a model-driven performance engineering method, SimuLizar, which supports software engineers to design self-adaptive information systems, annotate performance-relevant information, and predict these systems' scalability and elasticity properties.

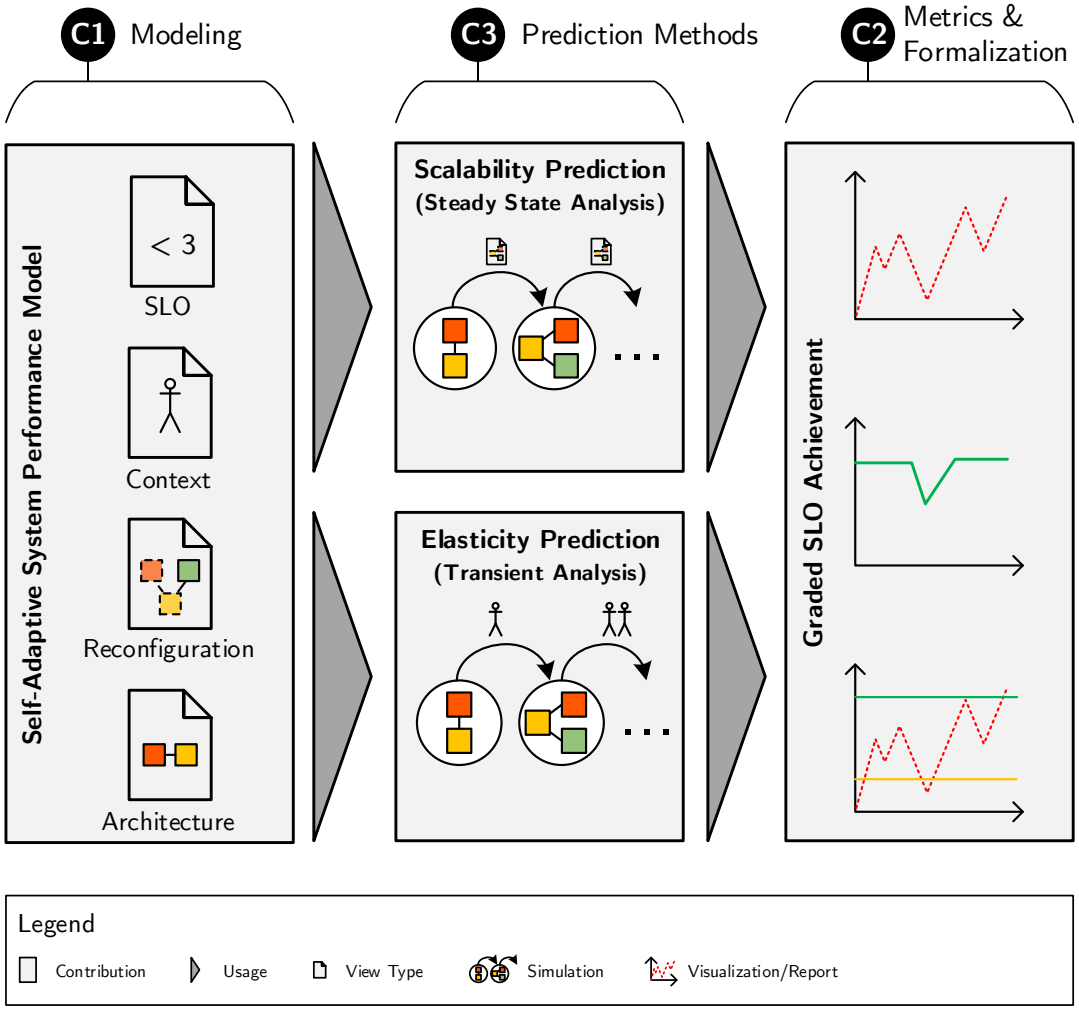


Figure 1.3.: Scientific contribution

Moreover, we provide a formalization and concrete metrics for the scalability and elasticity properties. We implement our model-driven method in a tool, SimuLizar Bench. Figure 1.3 illustrates the key contributions of this thesis:

C1 **Performance Modeling** We provide three viewpoints for modeling self-adaptive systems: A system type viewpoint, a run-time viewpoint, and a self-adaptation viewpoint. The viewpoints and their respective view types are implemented as meta models in the Eclipse Modeling Framework and build on the Palladio Component Model [BKR09]. We specifically extend the Palladio Component Model with three additional view types: a reconfiguration view type, which is used to specify architecture reconfigurations; a monitoring view type, in which monitors for self-awareness capabilities are specified; and a service level objective view type in which service level objectives are specified which are major drivers for self-adaptation.

C2 **Metrics & Formalization** We define metrics to quantify scalability and elasticity properties of self-adaptive systems. In contrast to other related work, the use case for these metrics is the assessment and comparison of self-adaptive system designs early at design-time. For the assessment of the scalability and elasticity of self-adaptive systems, we provide a formalization of service level objectives (SLOs) based on fuzzy logic [Zad65]. This formalization introduces a notion of graded SLO achievement and thus facilitates the decision-making between system design alternatives [Pla16].

C3 **Prediction Methods** We provide methods to predict scalability properties and elasticity properties of self-adaptive system designs at design-time. Both methods are defined based on our formalization of service level objectives and are evaluated regarding their applicability within a case study.

Based on the contributions listed above, we implemented SimuLizar Bench, a tool that extends Palladio Bench with our viewpoints for self-adaptive system performance modeling. We implemented scalability and elasticity prediction methods in the SimuLizar Bench by building on Palladio Bench's SimuCom simulation framework [BKR09].

We present the first contribution, C1, our self-adaptive system performance modeling approach in Chapter 4. Our scalability and elasticity metrics and prediction methods, i. e., Contribution C2 and Contribution C3, are presented in Chapter 5. The evaluations of each contribution can be found in their respective chapter. Finally, the implementation of our contributions in SimuLizar Bench are presented in Chapter 6.

1.4. Overview

The remainder of this thesis is structured as follows. In Chapter 2, we introduce basic terminology in the context of self-adaptive systems. In Chapter 3, we introduce the foundational concepts of software performance engineering, which builds the basis for the prediction methods that are presented in this thesis. In particular, we outline the characteristics of cloud computing and self-adaptive systems. Furthermore, we introduce model-driven engineering and view-based modeling as a basis for our performance modeling approach. Model-driven software performance engineering in general and the Palladio approach in particular are introduced as foundations for our model-driven scalability and elasticity prediction methods. We introduce our performance modeling approach for self-adaptive systems in Chapter 4. The approach is illustrated on the Znn.com example system and the models serve as input for the analysis methods that we present in the subsequent chapter. In Chapter 5, we present the elasticity and scalability prediction methods including their formalization, implementation, and limitations. Thereafter, in Chapter 6, we present SimuLizar Bench, as the implementation of our modeling approach and scalability and elasticity prediction methods. Finally, in Chapter 7 we conclude this thesis and point to directions for future work.

"Exploring the unknown requires tolerating uncertainty."

Brian Greene

2

Uncertainty, Imprecision, and Gradedness in Software Engineering

Contents

2.1. Definitions and Examples 18

2.2. Applications in Software Engineering 23

In this chapter, we introduce the first part of the foundations for this thesis. In the context of self-adaptive system engineering, the three terms *uncertainty*, *imprecision*, and *gradedness* play a major role. Hence, we provide definitions for all three terms and provide examples in which areas of software engineering the terms are relevant. Furthermore, we introduce related software engineering methods and concepts that specifically address uncertainty, imprecision, and gradedness in different software engineering phases.

2.1. Definitions and Examples

Fuzzy sets [Zad65] and fuzzy logic [KY95] are commonly used to model *uncertainty*, *imprecision*, and *gradedness*. In this section, we introduce the definitions of all three terms and provide examples in the context of our Znn.com system that we introduced in the previous chapter.

Definition 2.1 (Fuzzy Set) *Let X be a space of points (objects), with a generic element of X denoted by x. Thus, $X = \{x\}$.*
A fuzzy set A in X is characterized by a membership function $\mu_A(x)$ which associates with each point in X a real number in the interval $[0, 1]$, with the value of $\mu_A(x)$ at x representing the "grade of membership" of x in A. [Zad65]

A fuzzy set is a mathematical set that in contrast to naive sets [Hal60], or *crisp sets*, defines a graded membership for all elements within the set. Definition 2.1 formally defines fuzzy sets. According to this definition, a membership function $\mu_A(x)$ associates for each element x within a domain X, e. g., $x \in \mathbb{R}_0^+$, a real number in the interval $[0; 1]$ that represents the membership grade for x in fuzzy set A.

Figure 2.1 illustrates a membership function for a real number value domain, i. e., $x \in \mathbb{R}_0^+$. In the example, the real number values between 1.0 and 2.0 have a membership grade of 1.0. Values of x that are greater than 3.0 have a membership grade of 0.0, i. e., are no members of the fuzzy set A. Values of x in $(0.0, 1.0)$ and $(2.0, 3.0)$ have a graded membership in A as defined by the function $\mu_A(x)$.

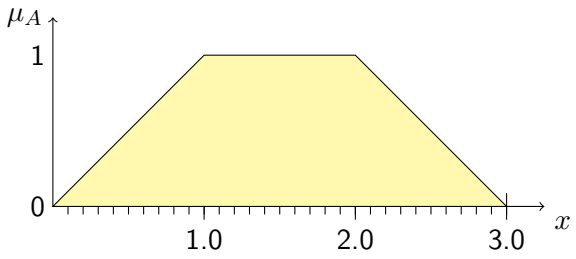


Figure 2.1.: Example membership function for a fuzzy set.

The membership grade of a variable x in a fuzzy set A can be interpreted in different ways. First, it can be interpreted in terms of *uncertainty*. That is, the membership grade defines a possibility that the variable x is contained in A. Second, the membership grade can be interpreted as a way to express *imprecision*. That is, the membership grade states that A is not precisely defined, such that a precise mapping of x is not possible. Finally, the membership grade can be interpreted as a *gradedness*. That is, the membership grade of x exactly reflects the grade of which x is member of A.

While fuzzy sets are well suited to model the concepts *uncertainty*, *imprecision*, as well as *gradedness*, they are interpreted differently as outlined above. All three concepts and their interpretation are useful in different contexts of software engineering. In the following, we give an overview of the three concepts and concrete applications in this thesis and in related software engineering approaches.

2.1.1. Uncertainty

In our context, we can describe uncertainty as a property of an information. An uncertain information is either imperfect or not (completely) known. We can distinguish two types of uncertainty: (1) aleatoric (or aleatory) uncertainty and (2) epistemic uncertainty.

"The word aleatory derives from the Latin *alea*, which means the rolling of dice. Thus, an aleatoric uncertainty is one that is presumed to be the intrinsic randomness of a phenomenon." [DD09] For example, the request rate for our Znn.com example can be considered as subject to aleatoric uncertainty because of the random behavior of

humans that cannot be known upfront. "The word epistemic derives from the Greek $\epsilon\pi\iota\sigma\tau\eta\mu\eta$ (episteme), which means knowledge. Thus, an epistemic uncertainty is one that is presumed as being caused by lack of knowledge (or data)." [DD09] An example of information that is subject to epistemic uncertainty is the usage context of the Znn.com system. The concrete parameters of a request in our Znn.com example, e. g., which news article is requested by a client, as well as the actual frequency of customer request cannot be known at design-time. This information can only be assumed by the software engineer.

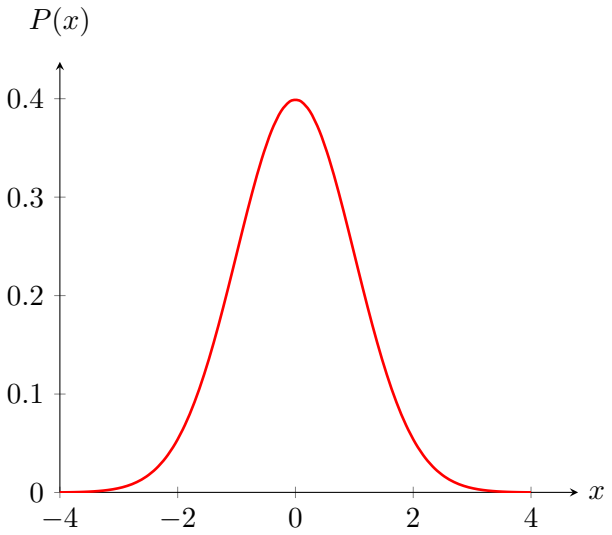


Figure 2.2.: Probability density function $P(x)$ for a standard normal distribution.

Uncertain information can be modeled either in terms of *probability* or in terms of *possibility*.

Probability is usually modeled via *random variables*. A random variable X is a function that maps from a probability space, e. g., set of observable events, to a measurable space, e. g., the set of real numbers $\mathbb{R}$. That is $X : \Omega \to E$, where Ω is a probability space and E is a measurable space. For example, X may define the number of heads for a random number of coin flips. The probability of a concrete event, e. g., "less than 3 heads", is then denoted with $P(X \leq 3)$. Continuous random variables can be defined with probability density functions and discrete random variable can be defined probability mass functions. The latter define a probability for each element in the set of (discrete)

events, e. g., $P(X = 3)$. A probability density function defines a probability that the random variable falls within a range of values, e. g., $P(0 \leq X \leq 3)$. Figure 2.2 illustrates the probability density function of a Gaussian distributed random variable. The area under the probability distribution function must be equal to 1.0, i. e., $\int_{-\infty}^{\infty} P(x) = 1$, because of the normalization condition defined in probability theory. That is, the sum of the probabilities of all values is 100%. [Sal07]

Possibility can be modeled with *fuzzy numbers*. A fuzzy number is characterized by a convex, normalized fuzzy set. Similar to probability, a possibility distribution function $\pi_X(x)$ defines the possibility that $X = u$, where X is a variable taking values in U and $u \in U$. For example, $\pi_X(3) = 1.0$ may be interpreted as: "It is completely possible (plausible) that X is 3". In this context, the possibility that $X = u$ is equal to $\mu(u)$, where $\mu(u)$ is the membership function that describes an event that is associated with the variable X, e. g., the possibility that it will be sunny for the next X days. For the membership function, a normalization condition is defined similar to the normalization condition in probability theory [Sal07], i. e., the supremum of this function must be equal to 1.0, i. e., $sup(\mu(x)) = 1$.

2.1.2. Imprecision

Imprecision is the lack of precision, for example caused by human interaction and errors or the ambiguity of language. In that sense, imprecise information is, like uncertain information, imperfect information. For example, in our Znn.com system, requirements could be imprecise. Typical examples of imprecise requirements would be "The system shall be fast" or "The operating costs shall be low". In general, imprecision often stems from a discrete representation, e. g., "fast", for a continuous variable, e. g., speed. Thus, imprecision may be reduced by choosing a different representation. However, this requires that a more precise representation exists and an exact value can be assessed.

Similar to uncertain information, the imprecision of information can be modeled with fuzzy sets. As illustrated in Figure 2.3, a fuzzy set can be used to describe the term "fast". The membership function μ_A defines the membership grade of speed values s in the fuzzy set A_{fast}, where $s \in \mathbb{R}_0^+$ and the unit of s is km/h. Intuitively, with this

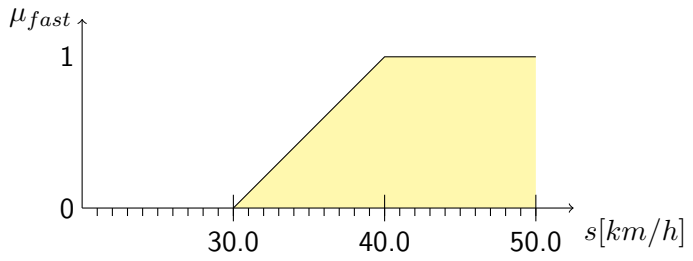


Figure 2.3.: Imprecise definition of "fast", modeled with a fuzzy set.

definition, speeds above $40km/h$ are considered as "fast", speeds between $30km/h$ and $40km/h$ are "more or less fast". Speeds below $30km/h$ are considered as "not fast".

2.1.3. Gradedness

Gradedness, in contrast to uncertainty and imprecision, describes no imperfect information. On the contrary, gradedness is a concept to model potentially ambiguous information in terms of a precisely defined mathematical object [Zad08]. For example, in this thesis, we define service level objectives (SLOs) with a notion of graded achievement. Thus, an SLO can be fully achieved or partly achieved to a certain grade. In contrast to our notion, in the traditional notion of SLOs, an SLO is either "achieved" or "not achieved". Thus, our SLO notion potentially reflects the preferences and requirements of the stakeholders more precisely.

Gradedness can be modeled with fuzzy sets as well. The membership grade of a value x in a fuzzy set A is then simply interpreted as the grade of x in A. For example, we can define a fuzzy set A_{mrt} that reflects a service level objective for an (imprecise) requirement "The mean response times shall be low" of our Znn.com system. The membership function μ_A then precisely defines to which grade this service level objective (and the requirement) is achieved at which mean response time values.

In Chapter 5, we introduce our formalization of *graded* SLO achievement and provide examples for our Znn.com system.

2.2. Applications in Software Engineering

In this section, we illustrate applications of uncertainty, imprecision, and gradedness in software engineering that are related to the concepts and methods that are presented in this thesis. The following subsections roughly follow the first three phases in software lifecycle [Bal11]: requirements engineering, design, and implementation. For each of the three phases, we highlight artifacts in which the concepts uncertainty, imprecision, or gradedness, as introduced in the previous section, are applied.

2.2.1. Requirements Engineering

A *requirement specification* is the artifact that results from the *requirement engineering* phase in the software lifecycle [Bal09]. The requirement specification specifies demanded properties, i. e., *requirements*, of the software system to be implemented. Requirements are typically elicited by *requirements engineers* from the software system's stakeholders, e. g., customers, marketing, management.

Requirements can be differentiated into *functional* and *non-functional* requirements. Functional requirements specify functions or services of a software system [Bal09]. An example functional requirement for our Znn.com system is: "The system shall be deliver news websites." Non-functional requirements, also called quality of service, describe aspects that often concern multiple functional requirements [Bal09], e. g., performance or operating costs. Often different non-functional requirements contradict each other [Bal09]. For example, operating costs and performance contradict each other when higher performance is only achievable with more (expensive) hardware resources. Hence, there is a trade-off between low operation costs and high performance.

RELAX Requirements Language

In classical requirements engineering, trade-offs between contradicting requirements have to be solved in the requirements specification, because requirements need to be

consistent [Bal09]. For this purpose, many different methods exist to solve trade-offs of contradicting requirements [vDL98].

However, the solution of a trade-off between contradicting non-functional requirements may not be completely possible or not wanted in the requirements engineering phase. For example, if contextual information of the software system, e. g., the number of customers, is not known a priori. In this case, it may be helpful to explicitly address this uncertainty in the requirements specification. Whittle et al. present with RELAX [WSB+10] a requirements language, that can be used to express requirements specification that consider uncertainty.

In contrast to traditional requirement specifications, in RELAX, a requirement engineer can distinguish between invariant requirements that a system has to meet at all time and non-invariant requirements that only need to be met in a best-effort fashion. The latter requirements are also called RELAX-ed requirements.

"A RELAX-ed requirement consists of two parts. The first part is a RELAX expression. In the second optional part, uncertainty factors in the environmental context can be specified in a declarative fashion. We formulate a RELAX-ed requirement for the operation costs of our Znn.com system as shown below. The RELAX keywords are denoted in capital letters. [BLB13]"

R3: The system SHALL keep the operation costs AS CLOSE AS POSSIBLE to 0.

ENV: number of customers; operation costs

MON: request rate; number of leased resources

REL: the more customer requests are delegated to leased resources, the higher are the operation costs

In the original requirements, that we presented in Section 1.1, we required the Znn.com system to maintain the operation costs below USD 5.00, see Requirement R2. This requirement contradicts Requirement R1 in which a mean response time of 2.0 seconds is specified. Relaxing Requirement R2, as shown above, enables the system to potentially fulfill both requirements Requirement R1 and Requirement R2 at once. Furthermore, the RELAX requirement describes uncertainty in the system's context using the ENV

keyword. In the RELAX requirement the number of customers and the actual operating costs may be unknown, i. e., uncertain. However, the MON and REL statements describe how this context can be monitored indirectly. A MON statement describes something that the system can monitor, e. g., in the example above the number of leased resources and the request rate. The REL statement describes the relation between the uncertain context and what can be monitored by the system. In this example, we can estimate the operating costs by monitoring the request rate and the number of leased resources.

The semantics of RELAX requirements are defined in terms of *fuzzy branching temporal logic (FBTL)* [MLL04]. FBTL is a temporal logic including concepts for uncertainty in time and logical predicates using fuzzy logic concepts. That is, with FBTL one can describe fuzzy states and fuzzy events. In fuzzy logic, variables have a gradual truth value ranging from 0 to 1 in contrast to Boolean logic where truth values can only be either false (0) or true (1). [BLB13] As introduced in Chapter 2, the basis for fuzzy logic are fuzzy sets.

A RELAX requirement can be gradually fulfilled, i. e., it can be fulfilled no at all, completely fulfilled, or fulfilled to any grade in between. In the example above, the requirement would be completely fulfilled if the operation costs were USD 0.00. This would only be the case if the system does not require (and lease) any resources at all. In other cases, in which the operation costs exceed USD 0.00, the RELAX requirement can only be gradually fulfilled.

In this thesis, we introduce the notion of a graded service level objective, achievement similar to the concept of RELAX-ed requirements that is also based on FBTL. Similar to RELAX-ed requirements, we do not address uncertainty with this notion but gradedness, such that we can precisely define when service level objectives are achieved and to which grade.

2.2.2. Design

In the *design* phase of the software lifecycle, a solution to the specified requirements is created. The creation of the solution, the design, includes the creation of a software

architecture by a *software architect*. The software architecture specifies the structure and behavior of a software system in the form of architecture elements, their behavior, their interaction, and physical deployment [Bal11].

Software architectures are usually not created from scratch, but software architects reuse well-established *architecture styles* or *architecture patterns* for known software system classes [Bal11]. Architecture styles are more abstract, define a software system class in terms of a common vocabulary, and constrain how elements in the architecture can be combined [SG96]. In contrast, architecture patterns specify a software architecture scheme that is applicable to general, recurring design problems [BHS07]. Architecture patterns can be refined to concrete architectures. For example, the *server-client* style is an architecture style and *three tier* is a well-established architecture pattern for the class of *web information systems*, like our Znn.com system. The three tier architecture pattern defines a software architecture with three tiers, i. e., a presentation tier, an application tier, and a data tier. The division of the architecture into tiers allows the physical deployment to different hardware nodes as well as a clear separation of the task for each tier.

Self-Adaptive Systems

The class of self-adaptive systems describes software systems that autonomously adapt their structure and behavior at run-time in response to changes in their context. Thus, self-adaptive systems can operate in *uncertain contexts*. The Znn.com system, for example, is a self-adaptive system that autonomously adapts to its usage contexts. The usage context of the Znn.com system is uncertain, i. e., its workload is not known in the design phase. Thus, a non-adaptive architecture may not completely satisfy the requirements in all possible usage contexts, e. g., when the system does not have sufficiently enough resources to maintain the required mean response time for a high workload.

The *MAPE-K* architecture style specifies a high-level architecture style for the class of *self-adaptive systems* [Mur04]. It is named after the four phases of a self-adaptation *feedback loop*: *monitor*, *analyze*, *plan*, and *execute*. The "K" in MAPE-K stands for the *knowledge* that is accessed during all four phases.

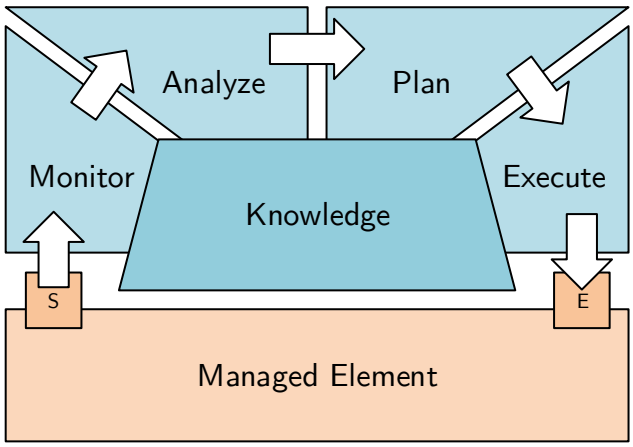


Figure 2.4.: MAPE-K feedback loop [Mur04].

Figure 2.4 illustrates the MAPE-K architecture style for self-adaptive systems. A self-adaptive system consists of two parts in the MAPE-K architecture style: a self-adaptation *feedback loop*, i. e., MAPE-K, and a *managed system*. The managed element is the actual software system that implements the business logic of the system, i. e., it implements a solution to the functional requirements. The managed element is monitored and adapted via *sensors* (S) and *effectors* (E). The MAPE-K feedback loop monitors the sensors, analyzes the monitored data, plans actions, and executes these actions via the effectors of the managed element.

In the context of software architecture, the actions that can be executed on the managed element are called *reconfigurations*. A reconfiguration transforms a specific software architecture instance, an *architecture configuration*, to another architecture configuration. For example, in our Znn.com system, a reconfiguration is the replication of the application tier, which includes the instantiation of a software component and its deployment to a hardware node. The set of possible reconfigurations of a self-adaptive system, i. e., the specification of the MAPE-K feedback loop, can be considered as an individual aspect of the overall system architecture. This aspect is also referred to as the *self-adaptation layer* of the system architecture.

The MAPE-K feedback loop adapts the system architecture autonomously in order to fulfill the system's requirements. Typically, requirements are reflected by service level

objectives (SLOs) at run-time. An SLO defines thresholds for quantifiable properties of a software system, e. g., maximum operating costs or minimum availability of the software system. Thus SLOs also define the borders for the autonomous behavior of a self-adaptive system. However, a self-adaptive system shall autonomously operate within these borders and autonomously solve trade-offs between contradicting requirements depending on its actual context.

Typically, a software architecture is specified from different viewpoints, which describe different aspects of the software architecture like structure in a static viewpoint, behavior in a dynamic viewpoint [GBB12]. In Chapter 4, we introduce a modeling approach that contains a dedicated self-adaptation viewpoint that addresses the specification of the self-adaptation layer.

Quality Assurance

Different aspects of software architecture are typically also modeled with *domain-specific modeling languages (DSMLs)*. A DSML allows software architects to model a specific aspect of a software architecture using the relevant concepts of the aspect the software architect is interested in. For example, a software performance DSML, can be used to specify performance-relevant aspects of a software architecture, such as available resources and resource demands.

DSMLs are often used as the basis of *quality assurance* of software architectures. The goal of quality assurance is to assess whether the implementation of the designed software architecture will fulfill the specified non-functional requirements and thus to verify design decisions. Hence, quality assurance of the software architecture should accompany the whole design phase and should not be applied at the end of the design phase only [Bal11]. Each quality aspect, e. g., performance, may require its own quality assurance method, such as *performance prediction*, and own DSMLs, like *software performance models*.

Quality assurance methods have to consider uncertainty since not all necessary information may be available at the time the method shall be applied. This especially applies in the context of self-adaptive systems since these systems are designed to work in un-

certain contexts. For example, the actual number of users and the resulting workload may not be known early in the design phase. In addition, information of the system itself may be uncertain in the design phase. For example, concrete resource demands are unknown in the design phase when no implementation exists. In quality assurance methods, like performance prediction, uncertain information is typically modeled using probability. The actual workload, that is required to predict performance metrics like response time, is then abstracted with a probability distribution that approximates the expected actual values.

We illustrate how uncertainty is addressed in quality assurance methods in more detail in Section 3.3 where we introduce the *software performance engineering* method.

2.2.3. Implementation

In the *implementation* phase of the software lifecycle, *software engineers* realize the software design that has been created in the design phase [Bal11]. Similar like software designs, the implementation is usually not created from scratch. Software engineers can use existing *platforms*, *software frameworks*, and *software components*, to implement the software system.

A platform provides basic resources and functionality to implement a software system. Typically, platforms are divided into hardware platforms and software platforms. However, the borders between hardware platforms and software platforms are not clearly defined. For example, the Amazon Web Services (AWS) is a platform that provides hardware resources, e. g., Amazon Elastic Computing Cloud (EC2) and Amazon Simple Storage Solution (S3), but also the software interfaces to access and manage the hardware resources. In contrast, the Eclipse platform is a hardware-independent software platform that provides various software frameworks for the implementation of software systems.

A software framework as well as a software component are reusable software artifacts that implements a certain functionality. Usually a software framework is a collection of software components that implements an architecture style or pattern. Both, a software framework and a software component, typically have defined *interfaces* that are used

to interact with other components of the software system [Bal11]. For example, the Eclipse Modeling Framework (EMF) is a software framework, which provides interfaces and components to implement domain-specific modeling languages.

Cloud Computing

The term *cloud computing* describes platforms that provide access to resources that can rapidly be leased and released with minimal management effort [MG11]. Four key characteristics of cloud computing platforms are (1) *autonomous resource leasing*, (2) *resource pooling*, (3) *elasticity*, and (4) *monitoring* [MG11]. That is, cloud computing platforms provide interfaces for software systems that support autonomous resource leasing from a resource pool. Monitoring interfaces allow cloud computing-based software systems to continuously monitor their resource consumption and thus provide the necessary information to decide whether resources shall be leased or released.

Cloud computing platforms can be categorized according what kind of platform and resources they offer. In general, we can distinguish three classes of cloud computing platforms: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [MG11]. However, the borders between these three classes are blurry. While SaaS platforms like Google AppEngine provide ready to use software systems, IaaS platforms like Amazon EC2 mainly provide virtualized hardware nodes, like virtual machines. Microsoft Azure, which provides virtual machines with the Windows operating system and interfaces to databases can be categorized as a PaaS platform [AFG+10].

In the context of this thesis, we consider cloud computing based software systems as self-adaptive systems, as introduced in Section 2.2.2, because of their ability to autonomously lease and release resources, e. g., virtual machines.

"If I have seen further, it is by standing on the shoulders of giants."

Isaac Newton

3

Software Performance Engineering Foundations

Contents

3.1. Model-Driven Software Engineering 32

3.2. Software Metrics and Predictions 35

3.3. Model-Driven Software Performance Engineering 37

In this chapter, we present the foundation for our model-driven method to predict scalability and elasticity of self-adaptive systems at design-time. In Section 3.1, we outline the key characteristics of model-driven software engineering, in which models are the central artifact in the software lifecycle. We show how view-based modeling supports model-driven engineering with the organization of software models into different viewpoints. In Section 3.2, we introduce basic concepts for software quality assurance at design-time, i. e., software metrics and predictions. In Section 3.3, we show how these basic concepts are applied to predict the software quality property performance at design-time using a model-driven prediction method. Finally, we introduce a concrete model-driven software performance engineering method, Palladio, which builds the basis for our model-driven prediction method that we present in Chapter 5.

3.1. Model-Driven Software Engineering

Model-driven software engineering (MDSE) is a software engineering paradigm in which the model is the central artifact of the complete software engineering process [Bec08]. The goals of MDSE are to increase development speed, improve software quality, and make the inherent complexity of software systems more manageable through abstraction [SVC06]. These goals are mainly achieved by automatic model transformations that are used to generate software engineering artifacts such as source code or analysis models from a central software architecture model [SVC06].

By using automatic model transformations, repetitive and error-prone tasks performed by the human actors are reduced and consequently less inconsistencies, mistakes, and errors are introduced within the artifacts of the software engineering process.

In this section, we specifically detail two major aspects of MDSE. First, we describe *view-based modeling* as a basic paradigm for the organization of models into viewpoints, view types, and views. Second, we describe *model transformations* as the central concept of model-driven software engineering.

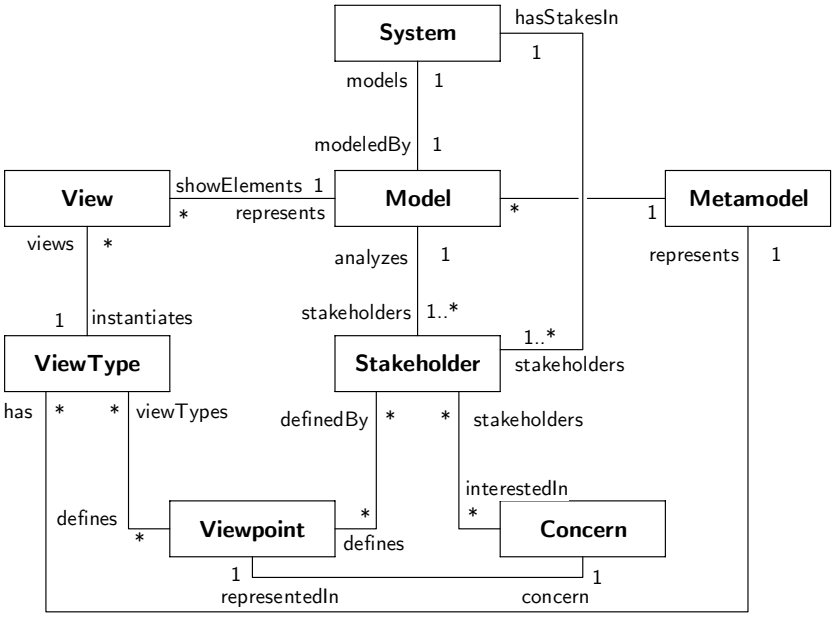


Figure 3.1.: Model that defines the taxonomy of view-based modeling [GBB12].

View-Based Modeling

With the term *view-based modeling*, Goldschmidt et al. [GBB12] describe a taxonomy for the organization of models in the context of MDSE.

The taxonomy is illustrated in Figure 3.1. In this taxonomy, a single *model* of a software *system* is represented by multiple *views*. A view is a projection of a model that shows a subset of the models elements. The model is an abstract representation of a system. The abstraction is specific for the model's purpose, i. e., elements or aspects of the system that do not contribute to the purpose of the model are omitted in the model. However, there should exist a homomorphism between the model and the system with respect to the model's purpose, i. e., the phenomenon that are observable on the system shall be observable on the model as well. The model is defined by a meta model. Since a single system usually has multiple *stakeholders*, which are interested in different *concerns* of the system, these concerns are reflected by different *viewpoints*. For example, performance is a concern of a software system's software engineer that is modeled in a dedicated viewpoint. Since different concerns are relevant to different stakeholders, the

concerns should be separated from each other. Each viewpoint defines multiple *view types* that in turn define the elements of a view. For example, in a software performance viewpoint, there may exist a view that models the resource demands of the methods in a software component.

Model Transformations

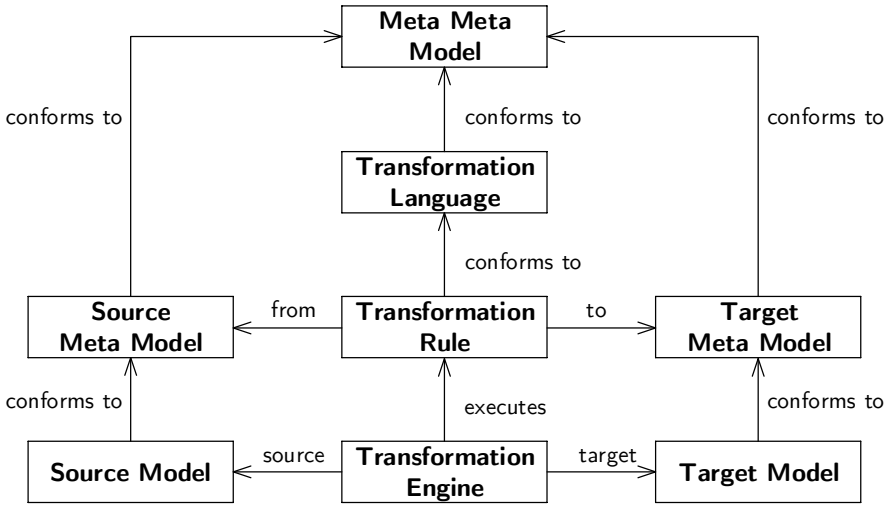


Figure 3.2.: Basic Concept of model transformation [DEP12].

Model transformations have a central role in MDSE. Model transformations are a set of *transformation rules* that specify how a *source model* can be transformed to a *target model*. As illustrated in Figure 3.2, the source model and the target model have to conform to their respective meta model. That is, the source model is an instance of the *source meta model* and the target model is an instance of the *target meta model*. The transformation rules are specified with *transformation languages* like the OMG Query/View/Transformation (QVT) [Obj16], Storydiagrams [FNTZ00], and Henshin [ABJ+10]. The transformation language and the source and target meta model conform to the same meta meta model, e. g., the OMG Meta Object Facility (MOF).

Commonly, model transformations are classified as either *model-to-model* (M2M) transformations or *model-to-text* (M2T) transformations. M2M transformations can be fur-

ther categorized as either *inplace* transformations or *outplace* transformations. In inplace M2M transformations, a single model is manipulated, i. e., source and target is the same model. In outplace M2M transformations, source and target are different models and are potentially instances of different modeling languages. In this thesis, we use inplace M2M transformations to model reconfigurations of a self-adaptive system, as we will describe in more detail in Chapter 4. Outplace M2M transformations are, for example, used in model-driven quality assessment methods to transform architecture models to analysis models, as we illustrate in Section 3.3. With M2T transformations, a source model is transformed into plain text. For example, the generation of source code is often realized as a M2T transformation. In the context of this thesis, we used a M2T transformation to generate code for a performance prototype for the evaluation of our prediction methods that we present in Chapter 5.

3.2. Software Metrics and Predictions

As already motivated, an early assessment of the non-functional requirements is crucial to prevent project budget overruns and to reduce time-to-market. The assessment of non-functional requirements requires *measuring* the corresponding non-functional properties of the software system. That is, the property has to be captured in a quantitative way.

Software metrics are used to quantify non-functional properties of software systems. Software metrics also play an important role in software engineering to monitor and control the quality of the software to be developed and to obtain the progress of the software's development. Requirements for non-functional properties, such as performance or costs, are usually specified in the requirements engineering phase of the software lifecycle.

In the context of software engineering, a *metric* can be defined as a precisely specified method to measure a property of a system. For example, response time is a performance metric for software systems. In this example, performance is the property of interest and response time is a metric that captures that property quantitatively.

Definition 3.1 (Metric) *A metric M is a function that maps an element of an (ordered) set V to a system S. [BF08]*
$M : S \to V$

In general, a metric is defined like in Definition 3.1. In the example above, the S is the performance of a software system and V is the set of positive real numbers, i. e., $\mathbb{R}_0^+$ with base unit seconds (s). The set V can also be a discrete set, like $bad, okay, good$. Thus, metrics can be used to classify, assess, and compare software systems [BF08].

Classification of Metrics

Metrics can be used for different purposes. With *analytical metrics* the properties of the system of interest are measured using models of the system [BF08]. Hence, analytical metrics are often used to predict properties of a system that is yet to be implemented. With *empirical* metrics, the properties are measured by observing the real system of interest [BF08]. This obviously requires that the system of interest is observable, i. e., the system has to be implemented and deployed.

Another dimension for the classification of metrics is defined by Reussner and Firus [RF08]. They distinguish between *basic metrics* and *dependent metrics*. A basic metric's dependencies on other metrics (or variables) are either not made explicit or no dependencies exist. For example, the software metric *lines of code* is a basic metric with no dependency on other metrics. In contrast to basic metrics, dependencies are explicitly defined for dependent metrics. The metrics we define in this thesis, scalability and elasticity, are both dependent metrics. In this thesis for example, we consider these metrics with respect to a system context metric, like the request rate. That is, the dependent metric (scalability) depends on another metric (request rate).

In this thesis, we define analytical, dependent metrics to assess performance properties of a self-adaptive systems, i. e., scalability and elasticity, in Chapter 5. We use a self-adaptive system performance model, that we introduce in Chapter 4, to simulate the system of interest and use our metrics to measure the scalability and elasticity properties. Thus, our metrics can be used to predict scalability and elasticity of a self-adaptive system early at design-time when no implementation of the system is available. In the

next section, we outline the model-driven software performance engineering that is used to obtain basic performance metrics at design-time.

3.3. Model-Driven Software Performance Engineering

Model-driven software performance engineering is a complement method to model-driven software engineering that introduces means to predict performance metrics early in the software lifecycle. For that purpose, the model of the software, the central artifact in MDSD, is enriched with performance-relevant information, such as resource demands, available resources, and the workload of the system. This enables to predict performance metrics of the software with *performance analysis tools*.

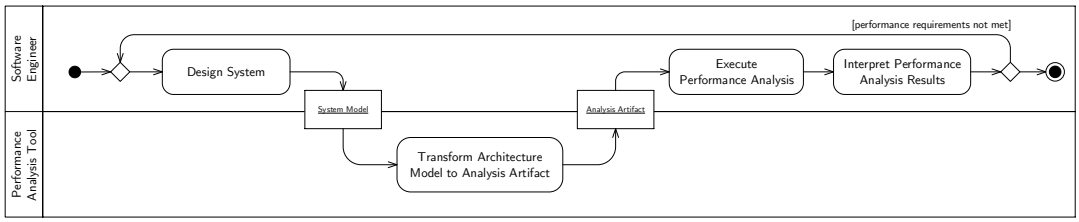
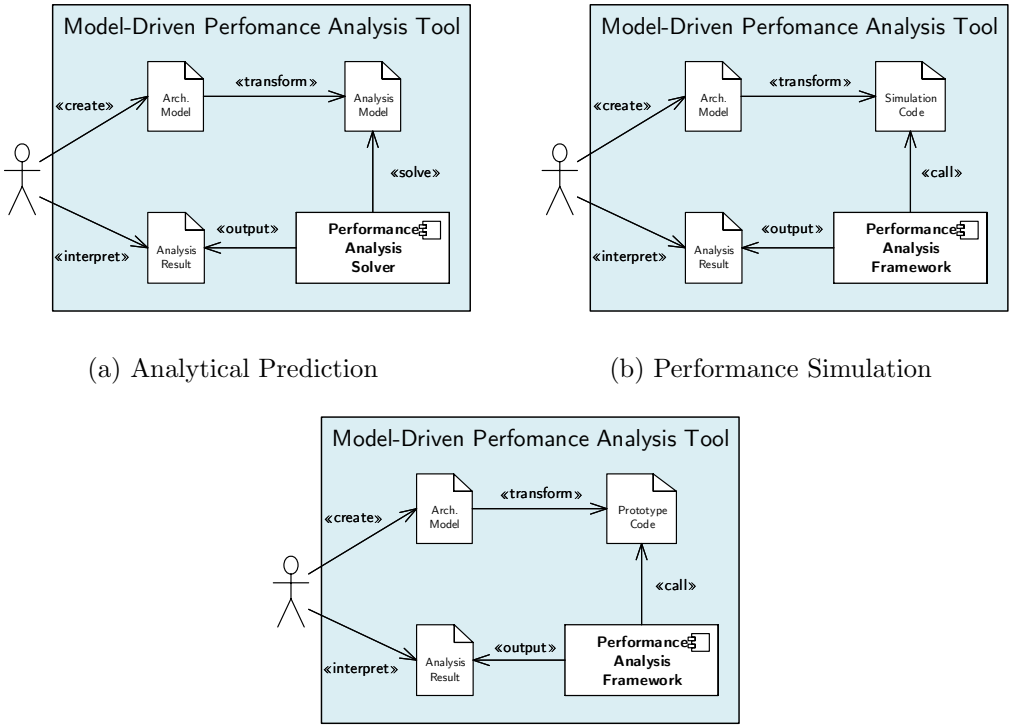


Figure 3.3.: Model-Driven Software Performance Engineering process.

Figure 3.3 illustrates a high-level process for model-driven software performance engineering. In the first step of this process, a *software engineer designs the software system*. The result of the design phase is a *system model* that includes the system's architecture with structural and behavioral aspects as well as performance-relevant information. In the second step, the model is *transformed to an analysis artifact* by a *performance analysis tool*. The concrete output of the model transformation, i. e., the type of analysis artifact, depends on the type of performance analysis tool. The analysis tool uses the generated artifact to *execute a performance analysis*. The *analysis results* of the execution are *interpreted* by the software engineer. If the metrics indicate that the requirements are met, the software engineering process can continue. Otherwise, the software engineer has to revise the software design, i. e., the system architecture, and rerun the analysis.



(a) Analytical Prediction

(b) Performance Simulation

(c) Performance Prototype

Figure 3.4.: Comparision of MDSPE analysis types.

Figure 3.4 illustrates three types of performance analysis tools: *solvers*, *simulations*, or *prototypes*. A performance solver predicts performance metrics via analytical solving of analysis models like layered queuing networks (LQN) [FAOW+09] on the basis of queuing theory [Mei58] and operational laws [DB78]. Performance simulation commonly uses the same analytical models like performance solvers, but instead of mathematical solving the models are simulated, i. e., the behavior of the system is symbolically executed. Thus, performance metrics of more realistic and hence also more complex models can be predicted [BKR09]. Finally, performance prototypes are software systems that implement the behavior of the specified software system in terms of performance, i. e., resource demands, but not semantically.

All three alternative analysis tool types have individual advantages and disadvantages: While solvers have the lowest execution time, they are restricted in the metrics and the accuracy of the metrics that can be predicted. Simulations are potentially more accurate than solvers but have higher execution times. Prototypes are most accurate in comparison to the other types but have the longest execution time and have highest hardware requirements and consequently highest costs [BKR09].

3.3.1. Palladio

Palladio is a model-driven software performance engineering method that provides a dedicated performance modeling language, the *Palladio Component Model (PCM)*, and a tool suite, *Palladio Bench*, which integrates different types of performance analysis tools.

Figure 3.5 provides an overview of the *partial models* in the *Palladio Component Model*, the model transformations with their respective output *analysis artifacts*, and the *analysis tools* implemented in *Palladio Bench*. We briefly summarize the building blocks of PCM and Palladio Bench in the following paragraphs.

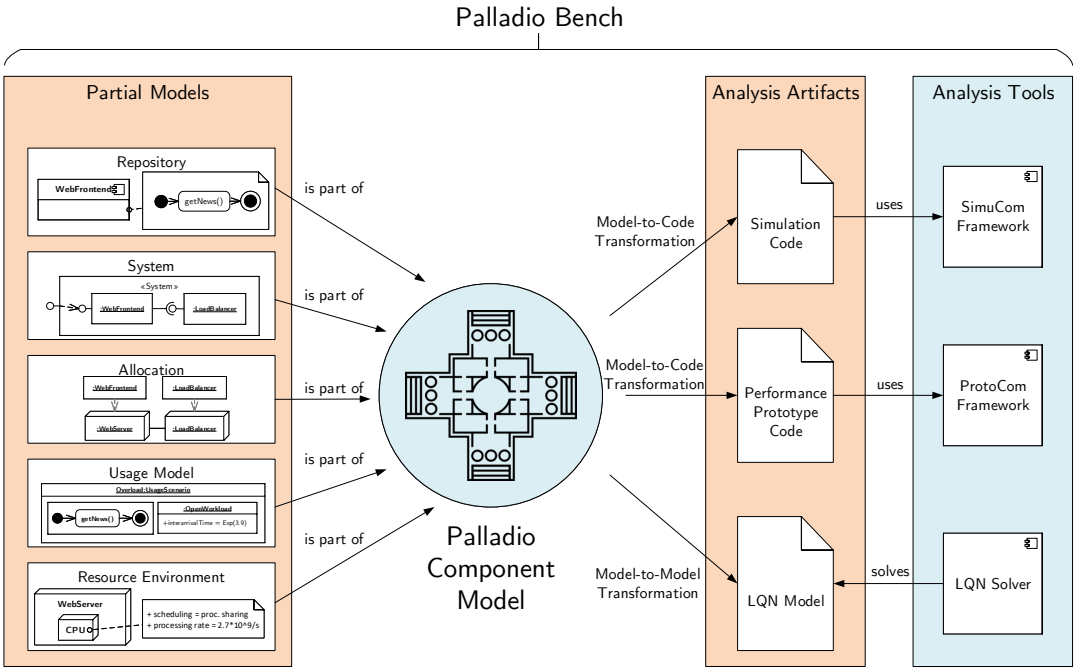


Figure 3.5.: Overview of PCM partial models and model transformations.

Palladio Component Model

As the name, Palladio Component Model, already implies, PCM is designed for *component-based software engineering* (CBSE) [SBW99, Szy02]. A PCM model is divided into five partial models that represent different views of a software system. The division into partial models enables to divide the modeling process according to the different roles in CBSE, i. e., *component developers*, *system architects*, *system deployers*, and *domain experts*. In the following, we describe the partial models in PCM and the roles involved in the performance modeling and analysis with Palladio.

Repository Model In the repository model, component developers specify component types, components, and interfaces. A component type is specified via its provided and required interfaces. In PCM, for components that conform to a component type, also a specification of the behavior has to be provided. Behavior specifications are called *service effect specifications* (SEFF) in PCM. The behavior specification is modeled as a flow of actions and can contain internal actions, ex-

ternal calls, loops, and forks. An internal action can contain performance-relevant information, i. e., resource demands. A SEFF that contains internal actions with resource demands is called *resource-demanding SEFF* (RD-SEFF). A resource demand is specified with *stochastic expressions* (StoEx). A stochastic expression is a mathematical term that can include random variables, like introduced in Section 2.1.

System Model Software architects compose a system from the components and interfaces, available in repository models. For this purpose, components that are specified in repository models, are instantiated and connected. Component instances are represented by so called *assembly contexts* in PCM.

Allocation Model Software deployers specify the allocation of the components instances within a system to hardware nodes.

Usage Model The usage model contains a specification of the user's behavior. This includes a specification user action flows, a characterization of the actual parameters, and a load characterization. The characterization of the actual parameters and the load is specified with stochastic expressions, just like for the parametric resource demand specifications in RD-SEFFs.

Resource Environment Model The resource environment model contains a specification of the available hardware resources, i. e., *resource containers* and *linking resources*. A resource container can represent either physical or virtual resources like a server or virtual machine. Resource containers can also contain active resources, like CPUs or hard disks. The capacity of an active resource is characterized by a scheduling strategy, e. g., processor sharing or first-in-first-out, and a processing rate. The processing rate is specified with stochastic expressions.

Palladio Bench

Palladio Bench integrates editors for the PCM with all three types of performance analysis tools: *LQN Solver* and *LINE solver* as performance solvers, *SimuCom* as a

performance simulation framework, and *ProtoCom* as a performance prototype framework.

LQN Solver and LINE solver PCM models can be transformed to LQN models via an automatic model transformation implemented in Palladio Bench. The resulting LQN model can be analyzed either with the LQN Solver [FAOW+09] or the LINE solver [PC13]. Both solvers are limited to the prediction of response times for open workloads and hence provide less comprehensive analysis results compared to the other two performance analysis tools in Palladio Bench.

SimuCom A performance simulation can be generated from PCM models via automatic model transformation. The generated simulation code uses the SimuCom framework, which is part of Palladio Bench. The SimuCom framework provides the basic functionality for performance simulations, such as the simulation of user requests, resource scheduling, and resource consumption. In Chapter 6, we present the implementation of our prediction method, which is based on the SimuCom framework.

ProtoCom Additional to performance simulations, performance prototypes can be generated with Palladio Bench as well. Similar to SimuCom, ProtoCom provides a framework that provides basic functionality for performance prototypes. With ProtoCom load can be generated and resources can be consumed on real hardware nodes. For the resource consumption, ProtoCom is calibrated on the hardware it is executed on in order to consume as much resources as specified in the PCM model.

"Meaningful architecture is a living, vibrant process of deliberation, design, and decision, not just documentation."

Grady Booch

4

Self-Adaptive System Performance Modeling

Contents

4.1. Scientific Contributions 45
4.2. Modeling Requirements 46
4.3. Related Work 54
4.4. Self-Adaptive System Performance Modeling Overview . . . 62
4.5. System Type Viewpoint 70
4.6. Run-Time Viewpoint 77
4.7. Self-Adaptation Viewpoint 86
4.8. Evaluation 99
4.9. Conclusion 107

Self-adaptive system architectures are more complex than non-adaptive system architectures due to the addition of a self-adaptation layer, as described in Section 2.2.2. The additional complexity also means a challenge for modeling self-adaptive systems for the purpose of performance analysis.

Research already addressed the specification of software architectures [Cle96, MT00] with special-purpose modeling languages, so called architecture description languages (ADLs). Software engineers use ADLs to model systems for the purpose of documentation, but also for analysis. However, no ADL that enables performance analysis of self-adaptive systems at design-time does exist yet.

On one hand, there exist some ADLs that also allow the specification of performance-relevant information within the architecture model [BDIS04, Koz10]. However, these ADLs focus on non-adaptive software architectures and thus do not provide means to specify the self-adaptation layer as an independent concern [Luc13]. Consequently, these modeling languages cannot serve as input for the prediction of relevant performance properties of self-adaptive systems, such as scalability and elasticity.

On the other hand, the specification of self-adaptive software architectures has been addressed by few approaches which either completely lack of means to specify performance metrics or over-simplify performance properties to single values that do not reflect performance properties in a reliable and reproducible way [BLB12].

Consequently, a modeling approach that enables the specification of self-adaptive system architectures for the purpose of analyzing these systems' performance properties, such as scalability and elasticity, is still missing.

Based on two independent sets of requirements [Bec11, Luc13] for self-adaptive system modeling approaches, we set up a comprehensive list of requirements for a modeling approach that also supports the specification of performance properties of a self-adaptive system. Subsequently, we surveyed literature and evaluated modeling approaches with our list of requirements. We identified the Palladio Component Model (PCM), as the approach that fulfills most of the requirements in comparison to the other modeling approaches. Hence, we extended PCM with the missing features that enable the specification of self-adaptive system architectures and the specification of performance prop-

erties of the self-adaptation layer. The result is part of SimuLizar, our model-driven approach for the prediction of scalability and elasticity that we present in this thesis.

SimuLizar's performance modeling approach enables software engineers to model self-adaptive system architectures annotated with performance-relevant information. The resulting architecture models serve as input for the prediction of scalability and elasticity properties early at design-time for systems that implement the modeled architecture.

In this chapter, we first outline our scientific contributions to performance modeling of self-adaptive systems in Section 4.1. Second, we specify requirements for performance modeling of self-adaptive systems in Section 4.2. Third, we discuss existing modeling languages that address the specification of self-adaptive systems regarding these requirements in Section 4.3. Fourth, in Section 4.4, we give a brief overview of our own performance modeling approach for self-adaptive systems that is part of SimuLizar. Next, we describe three viewpoints and their respective view types for our modeling approach in Section 4.5, Section 4.6, and Section 4.7. Subsequently, we describe the evaluation of our performance modeling approach with respect to our requirements in Section 4.8. Finally, we draw conclusions for the approach presented in this chapter in Section 4.9.

4.1. Scientific Contributions

The scientific contributions in this chapter can be summarized as follows:

- We provide requirements to model self-adaptive systems with the purpose to analyze the system's scalability and elasticity. We derive these requirements from initial work by S. Becker [Bec11] and related work by Luckey [Luc13] and refine the requirements from related work with more detail. The resulting requirements allow us to identify relevant modeling approaches to the first problem we formulated in Section 1.2, i. e., missing performance modeling approaches for self-adaptive systems.

- We systematically review modeling approaches based on our self-adaptive system performance modeling requirements to identify the most promising candidates to base our model-driven performance engineering approach on.
- The performance modeling approach we introduce in this chapter fulfills all of our requirements by extending the Palladio Component Model with self-adaptation specific viewpoints. Thus, we provide all necessary means to model a self-adaptive system for the purpose of analyzing its scalability and elasticity.
- In addition to the modeling approach, we introduce new roles and a modeling process to illustrate the practical application of our approach. The process extends Palladio's performance engineering process with additional roles and use cases.

4.2. Modeling Requirements

As motivated above, a performance modeling approach that enables the specification of self-adaptive system architectures and their performance properties is still missing. Hence, we formulate requirements that specify what a performance modeling approach needs to provide in order to enable modeling self-adaptive systems and assessing the scalability and elasticity of these systems at design-time.

We base our set of requirements on two independent sets of requirements for self-adaptive system modeling by S. Becker [Bec11] and Luckey [Luc13]. Both formulate requirements for modeling approaches that support the analysis of self-adaptive systems. S. Becker's requirements are more focused on the scope that needs to be provided by such a modeling approach. Luckey's requirements are more general and specify desirable properties of a modeling approach for self-adaptive systems. Consequently, we split our requirements into two sets. Our first requirements set defines *general requirements* for the implementation and application of modeling viewpoints for the purpose of analyzing the scalability and elasticity of self-adaptive systems. Our second requirements set defines the *scope of the viewpoints*.

We derive the first set, the *general requirements* for our viewpoints, from the requirements that are defined by Luckey [Luc13]. We derive the second set that defines the

scope of our viewpoints from the requirements presented by S. Becker [Bec11]. S. Becker specifies requirements for two viewpoints, a *system type viewpoint* and a *run-time viewpoint*. However, our requirements refine S. Becker's set of requirements. Specifically, we added a third viewpoint, the *self-adaptation viewpoint*. Additionally, we added the Requirement MR6 "System Resource Context" to address the specific role of resources in self-adaptive systems. Furthermore, we added Requirement MR9 "Service Level Objectives" and Requirement MR10 "Monitoring" in order to be able to specify service level objectives, which are important drivers for self-adaptation, and the required monitoring for self-awareness of the system. Finally, we also added the Requirement MR8 "System Context" as change in a self-adaptive system's context, e. g., increasing workload, is the main cause for a required self-adaptation.

4.2.1. General Requirements

In this first set of *general modeling requirements*, we specify requirements for desirable properties for a modeling approach that shall enable designing self-adaptive systems for the purpose of assessing the scalability and elasticity of these systems.

MR1 Design-Time: A suitable modeling approach supports a software engineer to model a self-adaptive system for the purpose to predict their scalability and elasticity properties early in the software engineering process. Hence, the model must support the specification of a self-adaptive system architecture and its relevant performance properties at design-time. The modeling approach shall furthermore allow the software engineers to successively refine the model to add further details, like detailed control flows and resource consumptions, as the software engineers determine these details.

MR2 Separation of Concerns: The modeling approach shall support software engineers to separate self-adaptation from other concerns, like the application logic. Concern-specific viewpoints and view types support this requirement. High-level architecture styles, like MAPE-K, help to achieve a separation of concerns. A suitable modeling approach provides at least the viewpoints for system type and run-time, as proposed by S. Becker [Bec11]. Furthermore, a role concept is re-

quired, that defines which roles in a software engineering process provide which artifact.

MR3 Analyzability: The modeling approach shall support software engineers to model all necessary aspects that are required to predict the scalability and elasticity of a self-adaptive system. Hence, a suitable modeling approach shall provide means to specify all information that is relevant to analyze the scalability and elasticity of the modeled self-adaptive system architecture.

MR4 Integrated Tool Chain: The modeling approach shall be integrated into a tool chain for modeling as well as the ability to assess scalability and elasticity properties of the modeled self-adaptive system. A suitable modeling approach provides editors that allow designing the system with all artifacts of the software architecture, integrated in one tool or tool chain.

4.2.2. Scope of the Viewpoints

In this second set of requirements, we specify the *scope of the viewpoints*, i. e., which view types shall be included in the viewpoints. Additional to the two viewpoints proposed by S. Becker [Bec11], we add a *reconfiguration viewpoint*. For each of the three viewpoints, we provide individual requirements to define their scope.

System Type Viewpoint The first viewpoint is the *system type viewpoint*. This viewpoint includes all view types that are used to specify elements of the self-adaptive system on type level, i. e., elements that are not specific to a single system but whose instances can occur multiple times in one system and can be reused in multiple different systems.

MR5 System Architecture Types: A self-adaptive system model shall enable to specify the system and its parts. Typically, in self-adaptive systems elements of a certain type can be instantiated and occur in multiple instances in the system, e. g., multiple instances of software components running on the application tier. Hence, the system type viewpoint shall provide a *system architecture types view*

type to specify the system architecture at type level. Type level elements of the system architecture are, for example, services or components and their interfaces.

MR5.1 The system architecture types view type shall enable the specification of structural system architecture element types such as *interfaces* and *component types*.

MR5.2 The system architecture types view shall enable the specification of performance-relevant behavior on type level, such as *parametric resource demand specifications* of methods, e. g., parametrized execution time on CPU, required disk access time, execution delay, etc. The specification shall allow specifying uncertain performance information as well.

MR6 System Resource Context: As already motivated in Section 2.2.3, self-adaptive systems that are based on cloud computing offers, can potentially use an unlimited set of resources, like virtual machines. These resources come in various types, capacities, and capabilities. Scalability and elasticity rely on the leasing and releasing of these resources. Hence, a self-adaptive system performance model shall enable to specify resource types in a *system resource context view type* in order to enable scalability and elasticity analysis.

MR6.1 The system resource context view type shall enable the specification of *active resource types* like CPUs or hard disks, i. e., resource types that actively process tasks.

MR6.2 The system resource context view type shall enable the specification of *passive resource types* like thread pools or connection pools, i. e., resource types that are used to limit parallel execution of tasks in active resources.

MR6.3 The system resource context view type shall enable the specification of the *capacity* and *capabilities* of resource types. This includes the specification of CPU frequency, number of CPU cores, and access rates of hard disk drives, for example.

Run-time Viewpoint In the second viewpoint, the *run-time viewpoint*, all view types are included that are used to specify elements of the self-adaptive system on instance level, i. e., elements that are specific to a concrete system architecture configuration.

MR7 Initial System Architecture: A self-adaptive system requires an initial system architecture configuration in which software components will be instantiated and deployed when the system is started. The run-time viewpoint shall contain a view type to specify this initial system architecture configuration. In this view type, instances of the system element types are composed to a complete and valid system architecture configuration and the deployment of this system architecture to hardware resources is specified.

MR7.1 The initial system architecture view type shall enable the specification of an *initial system architecture configuration*. A self-adaptive system architect must be able to instantiate system element types and to compose these instances to an initial system architecture configuration.

MR7.2 The initial system architecture view type shall enable the specification of an *initial system deployment*. A self-adaptive system architect must be able to specify the deployment of elements in the initial system architecture configuration to hardware resources.

MR8 System Usage Context: The run-time viewpoint shall contain a view type to specify usage contexts of the self-adaptive system, i. e., different *usage scenarios* in which the system operates. The usage scenarios define the workload the self-adaptive system has to process. As introduced in Section 3.3, workload is composed from the properties work and load. Work describes the effort needed to process a task. In our Znn.com example, work could be the effort of the Znn.com system to process a user request, prepare a news website, and send the website as a response. Load is the property that describes how frequent a specific work is done. For example, in the Znn.com system, the load can be defined by the number of requests sent to request a news website in one second.

MR8.1 The system usage context view type shall enable the specification of *work*, which is processed by the self-adaptive system. The type of requests sent

from users to the Znn.com system, for example, characterize the work. The type of requests is defined by the method that is called and the concrete parameter values that determine which data is processed. Since the concrete parameter values might change at run-time, the work may vary.

MR8.2 The system usage context view type shall enable the specification of *load*, e. g., the frequency of requests (with a certain work to be processed). Additional to the specification of the frequency the system usage context view shall enable to specify whether the workload is an open workload, i. e., independent from prior workload, or closed workload, i. e., dependent from prior workload. The load in our Znn.com system is an open workload, since the occurrence of a user request is independent from previous requests. However, the frequency of user requests might vary for different scenarios, e. g., depending on time, weather, political situation, or other factors that influence the users' interest in news.

MR8.3 The system usage context view type shall enable the specification of *time-dependent variation* in the work and load of the system. That is, additionally to the scenarios in which a fixed workload is defined, it shall be possible to specify how the workload varies over time.

Self-Adaptation Viewpoint In the third viewpoint, the *self-adaptation viewpoint*, all view types are included that are used to specify the self-adaptation of a system. This viewpoint is orthogonal to the first two viewpoints and may reference elements on type level as well as on instance level.

MR9 Service Level Objectives: The self-adaptation viewpoint shall contain a view type to specify the *service level objectives* of the system. SLOs are important drivers for self-adaptation, because they define quantitative thresholds in which a self-adaptive system should operate. Self-adaptation is required whenever these thresholds are exceeded or impend to exceed. Non-functional quality properties of software systems are often in conflict with each other, e. g., response time versus operating costs. Hence, it shall be possible to specify soft thresholds, i. e.,

desired limits, and also hard threshold, i. e., required limits. These soft and hard thresholds define the corridor for self-adaptation, as described in Section 2.2.2.

MR9.1 The service level objectives view type shall enable the specification of *soft thresholds* for a previously defined monitoring specification, e. g., 2 seconds mean response time. For example, a software engineer shall be able to specify, that a mean response time of greater than 2 seconds is not desired but does not violate the service level objectives. In this case, we call the 2 seconds threshold a soft threshold.

MR9.2 The service level objectives view type shall enable the specification of *hard thresholds* for a previously defined monitoring specification, e. g., 3 seconds mean response time. For example, a software engineer shall be able to specify, that a mean response time of greater than 3 seconds is not acceptable and will violate the service level objectives. In this case, the 3 seconds are called a hard threshold.

MR10 Monitoring Specification: Monitoring is a prerequisite for self-awareness and self-adaptation, as described in Section 2.2.2. Hence, the self-adaptation viewpoint shall contain a monitoring view type to specify which elements of the system and its environment have to be monitored, what metric needs to be monitored, how often it shall be monitored, and how the measured data shall be aggregated.

MR10.1 The monitoring view type shall enable the specification of the *location type* of a system type element or the *concrete location* of an element instance in the system to be monitored, i. e., the measuring point. For example, to monitor the overall response time of our Znn.com system, a measuring point must be located directly at the method that the user calls to retrieve the news site.

MR10.2 The monitoring view type shall enable the specification of the *metric* to be monitored, e. g., response time or utilization.

MR10.3 The monitoring view type shall enable the specification of *frequency* for the monitoring, e. g., every minute, every hour, etc.

MR10.4 The monitoring view type shall enable the specification of the *aggregation* method of the measurements, e. g., aggregation in fixed batches or sliding windows. Furthermore, the specification of the aggregation function must be supported, e. g., conjunctive aggregations like minimum, disjunctive aggregations like maximum, or averaging aggregations like arithmetic mean [TN07].

MR11 Architecture reconfigurations: The self-adaptation viewpoint shall contain a view type to specify *architecture reconfigurations*. An architecture reconfiguration describes the transition from one valid software architecture configuration to another valid software architecture configuration. The execution of this reconfiguration shall only be triggered, if a reconfiguration is required, i. e., if all preconditions for the reconfiguration hold.

MR11.1 The architecture reconfiguration view type shall enable the specification of reconfiguration rules. For example, leasing additional resources like virtual machines, reallocating components from one resource to another, or the substitution of one component type by another component type are possible architecture reconfigurations in the Znn.com system. All these reconfigurations describe a transition rule from a specific source architecture configuration to a specific target architecture configuration.

MR11.2 The architecture reconfiguration view type shall enable the specification of *resource demands* required for the execution of a reconfiguration rule, e. g., required execution time, execution delay, etc.

MR11.3 The architecture reconfiguration view type shall enable the specification of system configuration and context *preconditions* for reconfiguration rules. For example, in our Znn.com example, we like to specify a reconfiguration rule that adds an additional virtual machine to the application tier only if the precondition holds that the mean response time is greater than 3 seconds and that the number of virtual machines in the application tier is less than 3.

4.3. Related Work

Modeling approaches that are candidates to fulfill our modeling requirements can be found in three research areas that are relevant for self-adaptive system performance modeling. As illustrated in Figure 4.1, the research areas are *self-adaptive systems (SAS)*, *model-driven software engineering (MDSE)*, and *software performance engineering (SPE)*. We applied the guidelines for systematic literature reviews by Kitchenham [KDJ04, KC07] to find performance modeling approaches in the intersection of the three research areas that fulfill our modeling requirements.

The survey question, which guided our systematic literature review was: *Which of the existing approaches fulfills the most of our modeling method requirements?* We conducted the literature survey in the time from October 2011 to April 2016 with the search engine Google Scholar [Goo16]. We selected related work, according to Kitchenham's guideline in two steps. First, we included all papers that can be found on Google Scholar by searching for a combination of our predefined keywords. The keywords were combinations of the three research areas "model-driven (software) engineering", "self-adaptive systems", and "(software) performance engineering", with and without the word "software" as indicated by the brackets. Second, we scanned the abstracts of all results and filtered out all papers that did not match our acceptance criteria. We accepted all papers that indicated that the paper presents a modeling approach for self-adaptive systems with the purpose of analyzing the model regarding quality properties.

Based on the requirements MR1 to MR11 and our findings in our initial survey [BLB12], we have created a feature model, see Figure 4.2, to specify required features for a self-adaptive system performance modeling approach and to answer the survey questions of our literature review. The feature model also helps to classify the surveyed approaches. The mandatory features in our feature model are the features that occur as requirements for our self-adaptive system modeling approach in the previous section. All other features that are not listed as requirements in the previous section, are optional. Thus, any self-adaptive system modeling approach that completely fulfills our modeling requirements is also a valid configuration for the presented feature model. We highlighted the configuration of our own performance modeling approach, we present in this thesis,

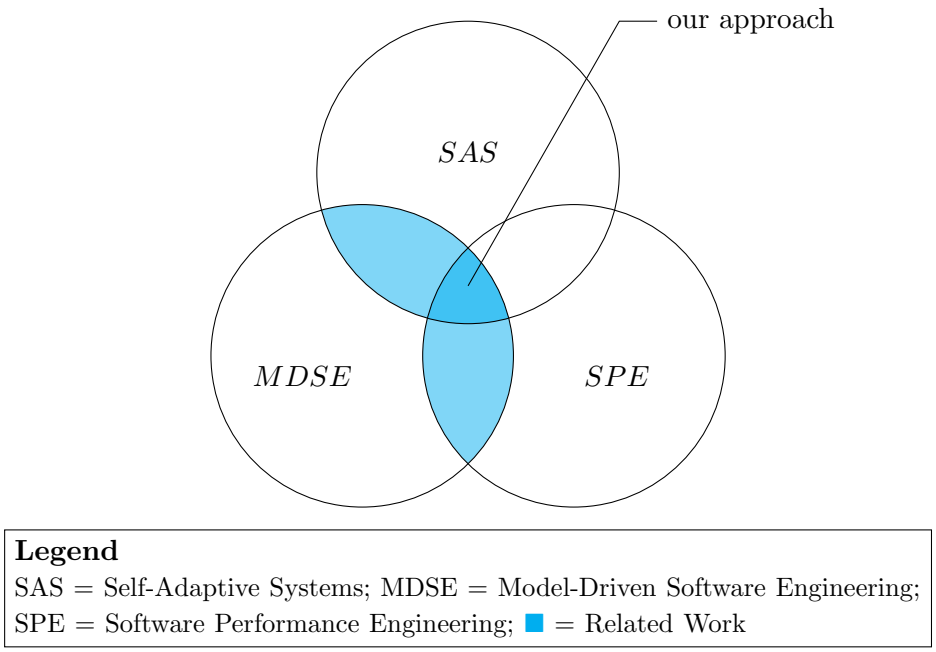


Figure 4.1.: Venn diagram of related work.

by coloring the selected features in gray in the feature model. We do not discuss related approaches that are based on the modeling approach that we present in this thesis. This follow-up work is discussed in Section 7.2.

Figure 4.2 shows that we classify self-adaptive system modeling approaches according to the target system class, the applicability of the approach, and the approach's modeling paradigm, i. e., , whether the approach is model-driven or model-based. First, we briefly summarize the targeted system classes and the applicability of the approaches. Second, we analyze whether the approaches fulfill our requirements MR1 to MR11 in more detail. We identify which features are already provided by existing modeling approaches and which features we have to add in our approach in order to fulfill all requirements.

The self-adaptive system modeling approaches we surveyed target either one of two different architectures: component-based architectures or service-oriented architectures (SOA). The majority of modeling approaches are targeted at component-based systems. Only the Adapt Case Modeling Language (ACML) [Luc13] and D-KLAPER [PPMMG10] target service-oriented architectures. Within the system class,

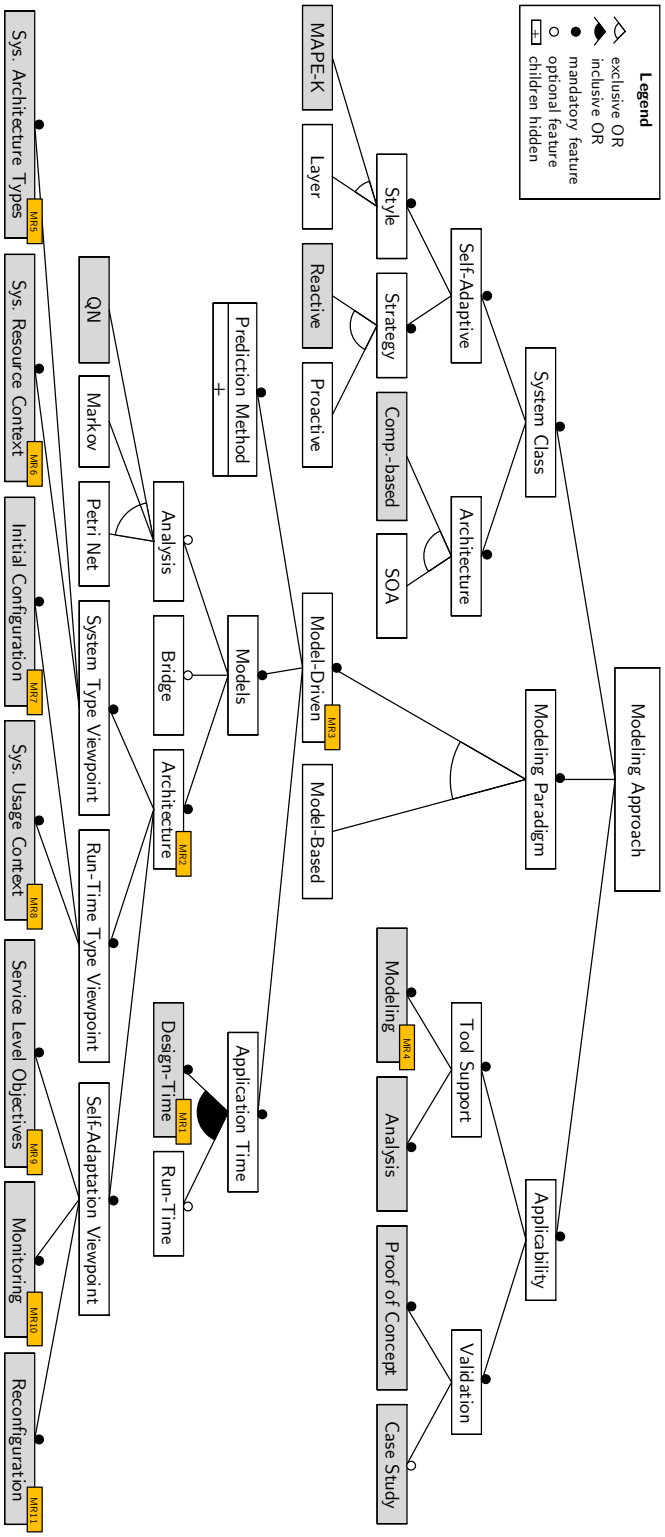


Figure 4.2.: Feature diagram for classification of related work.

we distinguish the self-adaptation style and strategy. The most common self-adaptation style is MAPE-K, as introduced in Section 2.2.2. Hence, also the majority of the surveyed approaches implement the MAPE-K style for modeling self-adaptation. However, for four of the approaches [FS09, ZC06, BKR09, PPMMG10] a self-adaptation style like MAPE-K does not apply because these approaches either do not oblige the software engineer to use a certain self-adaptation style or not fully support the specification of self-adaptation at all. The strategy when a self-adaptation is triggered can either be proactive or reactive, i. e., self-adaptations are either before SLOs are potentially missed or only after a SLO is violated. Except for the Descartes Modeling Language (DML) [HWBK15], which targets proactive self-adaptations, all other modeling approaches target reactive self-adaptation.

Regarding the applicability of the approaches, we found that all of the approaches at least provide proof-of-concept implementations. Only the Palladio Component Model (PCM) [BKR09] and DML [HWBK15] document more mature case study validations, in which they evaluate their modeling approaches. However, only the DML case study is in the context of self-adaptive systems modeling. The tool support for the modeling approaches is mixed. Only four of the approaches document modeling tools in publications. Two approaches mention the existence of tool support in the publications, but do not provide more details. No tool support is mentioned for the rest of the approaches.

Table 4.1 provides an overview of the fulfillment of the surveyed self-adaptive system modeling approaches regarding our requirements MR1 to MR11. From the table, we can first observe that none of the surveyed approaches fully fulfills our set of general modeling requirements nor our requirements specific for self-adaptation modeling viewpoints. However, we identified the Palladio Component Model as the best candidate to build our work on, because it fulfilled our requirements best since it already provides a good modeling approach for non-adaptive systems. The PCM also is available as an open source tool and is well documented. Thus, it provides a good basis for our own self-adaptive system performance modeling approach.

The first requirement MR1 “Design-Time” is fulfilled at least partially by all approaches with the exception of the DML [HWBK15]. This language is focused on run-time per-

Table 4.1.: Feature Configurations of Related Modeling Approaches

	General				System Type VP		Run-time VP		Self-Adaptation VP		
Approach	**MR1**	**MR2**	**MR3**	**MR4**	**MR5**	**MR6**	**MR7**	**MR8**	**MR9**	**MR10**	**MR11**
ACML [Luc13]	✓	✓	○	○	○	×	✓[1]	×	×	✓	○[1]
CLs [HGB10]	✓	×	×	?	○[1,2]	×	✓[1]	×	×	✓	○
D-KLAPER [PPMMG10]	✓	✓	✓	×	✓	✓	✓	○	×	×	○
DML [HWBK15]	×	✓	✓	○	✓	✓	○	✓	○	✓	○
Fleurey [FS09]	✓	✓	×	?	×	×	×	×	○	×	○
EUREMA [VG14]	○	✓	×	×	○[1,2]	×	×	×	×	×	✓
MUML [Gie07]	✓	○	○	✓	○	○	✓	×	×	×	○
PCM [BKR09]	✓	○	✓	✓	✓	✓	✓	○	×	○	×
Stitch [CG12]	✓	✓	×	×	×	×	×	×	×	×	✓
Zhang [ZC06]	○	×	×	×	×	×	×	×	×	×	✓

✓ = requirement fullfilled; × = requirement not fulfilled; ○ = requirement partly fulfilled;
? = unknown; - = does not apply; [1] using the UML; [2] using the MARTE profile

formance prediction and hence on run-time models. However, support for self-adaptive software architecture modeling at design-time is not provided by DML.

Requirement MR2 "Separation of Concerns" is fulfilled by the majority of the surveyed modeling approaches. However, the Control Loops (CLs) approach [HGB10] and Zhang's approach [ZC06] do not provide means to separate the self-adaptation concerns from the application logic in their modeling approaches. The PCM [BKR09] provides different view types for the specification of different aspects of the software architecture. However, view types to specify the self-adaptation concern are not provided. Hence, the requirement MR2 is only fulfilled partially by PCM.

The "Analyzability" requirement MR3 is, at least partially, fulfilled by half of the surveyed approaches. ACML [Luc13], D-KLAPER [PPMMG10], DML [HWBK15], MUML [Gie07], and PCM [BKR09] provide means to annotate relevant information for scalability and elasticity predictions within their models. However, the possible annotations for ACML and MUML are of limited use for scalability and elasticity predictions, since their resource demand annotations are either limited to either fixed scalar values or limited to worst-case execution times. The specification of uncertain resource demands, e. g., with probability distribution functions, is not supported in ACML and MUML.

Only two of the surveyed approaches provide an integrated tool for modeling analyzable self-adaptive system architectures, i. e., fulfill requirement MR4 "Integrated Tool Chain". MUML [Gie07] and PCM [BKR09] provide complete modeling tool suites that also support analysis. However, both tools do not provide a scalability and elasticity analysis. ACML [Luc13] and DML [HWBK15] provide modeling tools, but do not integrate analysis of non-functional properties like scalability and elasticity at all or not at design-time.

There are three approaches that support the full system type viewpoint as defined by our requirements, i. e., requirements MR5 "System Architecture Types" and MR6 "System Resource Context". DML [HWBK15], D-KLAPER [PPMMG10], and PCM [BKR09] fulfill both requirements for this viewpoint. ACML [Luc13], CLs [HGB10], and MUML [Gie07] provide partial support for at least one aspect of the system type viewpoint requirements. Requirement MR5 "System Architecture Types" is supported by

few of the approaches. In most of the approaches, (uncertain) performance information cannot be annotated within system architecture types. D-KLAPER, DML [HWBK15], and PCM [BKR09] are the exceptions here, as they allow performance information annotations like resource demands. These three approaches also provide means to model resources, as specified by requirement MR6 "System Resource Context".

The specification of the system's context, i. e., MR8, is only possible with two of the surveyed modeling approaches, i. e., DML and PCM. In D-KLAPER, the system's context is implicitly modeled within the resource consumption, i. e., the resource consumption is not parametrized but is modeled for one specific system context.

The PCM [BKR09] is the only surveyed modeling approach that completely fulfills the requirements for our run-time viewpoint defined by our requirements MR7 "Initial System Architecture Configuration" and MR8 "System Usage Context". The other approaches either do not fully support modeling resources, e. g., ACML [Luc13], D-KLAPER [PPMMG10], CLs [HGB10], and MUML [Gie07], or do not fully support modeling an initial system architecture configuration.

The three requirements MR9 "Service Level Objectives", MR10 "Monitoring Specification", and MR11 "Architecture Reconfigurations" that define our self-adaptation viewpoint are not completely fulfilled by any of the surveyed approaches.

Modeling service level objectives, i. e., MR9, is even not completely supported by a single approach of the surveyed approaches. The approaches that support modeling SLOs do not provide support to model soft thresholds, which is, however, relevant to distinguish a desired and required behavior of a self-adaptive system, as explained in Section 2.2.2.

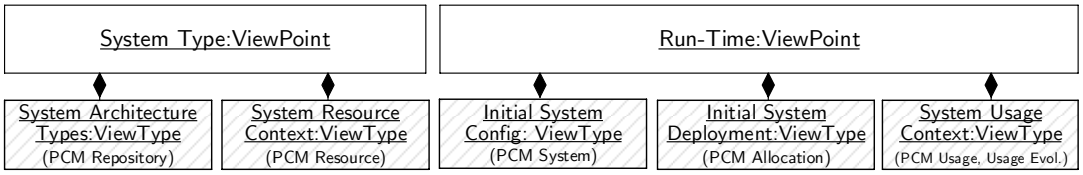
The requirement MR10 "Monitoring Specification", is only fulfilled by ACML [Luc13], CLs [HGB10], and DML [HWBK15]. All other modeling approaches do not allow the specification of monitors for the self-adaptation concern.

Interestingly, the approaches that fully fulfill our requirement MR7 "Initial System Architecture Configuration" do not completely fulfill our requirement MR11 "Architecture Reconfigurations" and vice versa. We assume that this is the case because the modeling

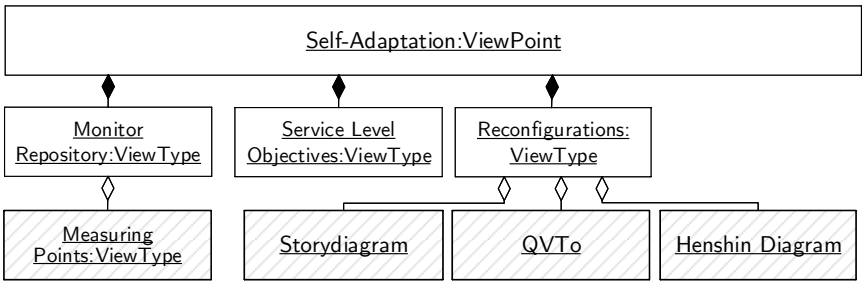
approaches are focused either on modeling analyzable software architectures or on modeling self-adaptation – but not both at once. For example, D-KLAPER [PPMMG10], MUML [Gie07], and PCM [BKR09] provide all very good support to specify an initial (non-adaptive) software system architecture configuration but no full support to specify architecture reconfigurations as defined by our requirement MR11. On contrast, EUREMA [VG12], Stitch [CG12], Zhang [ZC06] provide good support to model reconfiguration rules and preconditions but do not provide any support to model an initial software system architecture configuration.

In conclusion, we can summarize that none of the surveyed approaches completely fulfills our requirements. However, we identify the Palladio Component Model as the best candidate to build our work on. First, PCM scores best with completely fulfilling three of our general modeling requirements and fulfilling one of our general modeling requirements at least partly. Second, PCM also fulfills 3 out of 7 viewpoints requirements completely and two of the viewpoints requirements partly. Finally, the PCM meta model source is freely available, well-documented, and supported by an active community [Kar16a]. The second best candidate is DML, which is originally based on the concepts of PCM as well, but is focused on run-time performance predictions and hence does not provide design-time focused modeling tools.

Consequently, we selected PCM as the basis for our own self-adaptive system performance modeling approach. In order to completely fulfill our requirements for self-adaptive system performance modeling approach, PCM requires to additionally fulfill requirements MR9 “Service Level Objectives”, MR10 “Monitoring”, and MR11 “Architecture Reconfigurations”, i. e., the self-adaptation viewpoint, and the requirement MR8 “System Usage Context”. Hence, we extend PCM with a self-adaptation viewpoint and integrated the system usage context models from DML to fully support the modeling of variable system usage context. Additionally, MR2 “Separation of Concerns” is only partly fulfilled by PCM, since PCM does not provide distinct viewpoint for the self-adaptation concern yet. We provide an extended role concept and modeling process that extends PCM’s role concept and modeling process to also fulfill this requirement. Our extensions to PCM are part of SimuLizar, our model-driven engineering method for self-adaptive systems. We present SimuLizar’s complete self-adaptive system per-



(a) System Type Viewpoint and Run-Time Viewpoint



(b) Self-Adaptation Viewpoint

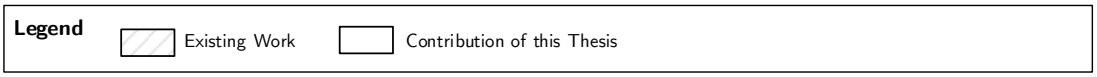


Figure 4.3.: Viewpoints in SimuLizar's self-adaptive sys. perf. modeling approach

formance modeling approach in detail in the remainder of this chapter. We also present which parts we reused from PCM and detail our extensions to PCM.

4.4. Self-Adaptive System Performance Modeling Overview

As motivated in the previous section, we selected the Palladio Component Model (PCM) as a basis for our self-adaptive system performance modeling approach in SimuLizar. In this section, we provide an overview of the viewpoints and the respective view types by explaining our extensions to the original Palladio Component Model. Additionally, we discuss additional and modified roles in our self-adaptive system performance modeling approach in comparison to Palladio's modeling roles, as introduced in Section 3.3. Finally, we describe how the roles are involved in the modeling process as part of the software engineering process.

4.4.1. Modeling Viewpoints

In Figure 4.3, our self-adaptive system performance modeling approach with its three viewpoints, the system type viewpoint, the run-time viewpoint, and the self-adaptation viewpoint, is illustrated. The white elements in the figure depict contributions of this work, the crosshatched grey elements are reused from PCM.

The system type viewpoint consists of two view types, which are reused from PCM. We reuse the PCM repository view type to specify system architecture types, i. e., component types and interfaces, of the self-adaptive system. Like shown in Table 4.2, elements that are specified in the system type viewpoint are on type level, e. g., components and interfaces. These elements have to be instantiated for a concrete system architecture configuration.

Our run-time viewpoint consists of another three view types. The PCM system view type is used for the specification of the initial system architecture configuration. The initial system deployment can be specified using the PCM allocation view type and the system usage context can be specified using the PCM usage view type. The variation of the workload in the system usage context can be modeled with Palladio's usage evolution [BSL16] and the Descartes Load Intensity Model (DLIM) [KHK+17]. The run-time viewpoint is a static viewpoint on an initial, concrete system architecture configuration.

In this viewpoint, elements of the system type viewpoint are instantiated, but no rules for the reconfiguration of this system configuration are provided. Like shown in Table 4.2 the view types in the system type viewpoint are used to model a concrete system configuration including its allocation and concrete usage scenarios. Thus, the run-time viewpoint reflects a static system architecture configuration, like in PCM.

Finally, the new self-adaptation viewpoint consists of three view types. First, it contains a monitoring view type to specify monitors for the self-adaptive system. The monitoring view type is an extension of the PCM measuring points view type. Second, the service level objectives view type is used to specify service level objectives with hard thresholds and soft thresholds. Finally, reconfigurations are specified in the Reconfigurations view type. Like shown in Table 4.2, in the self-adaptation viewpoint elements from the system type viewpoint and run-time viewpoint can be referenced. For example, monitors can

Table 4.2.: Elements in the Modeling Viewpoints

View Type	Top-Level Elements	Referenced View Types
System Type Viewpoint		
System Architecture Types	CompositeComponent, BasicComponent, Interface	-
System Resource Context	ResourceContainer, LinkingResource	-
Run-Time Viewpoint		
Initial System Architecture Config.	System	System Architecture Types
Initial System Deployment	AllocationContext	System Resource Context, Initial System Architecture Configuration
System Usage Context	UsageScenario, UsageEvolution	Initial System Architecture Config.
Self-Adaptation Viewpoint		
Service Level Objectives	ServiceLevelObjective	-
Monitor Repository	Monitor	System Architecture Types, System Resource Context, Initial System Architecture Configuration
Reconfigurations	Reconfiguration	Initial System Deployment

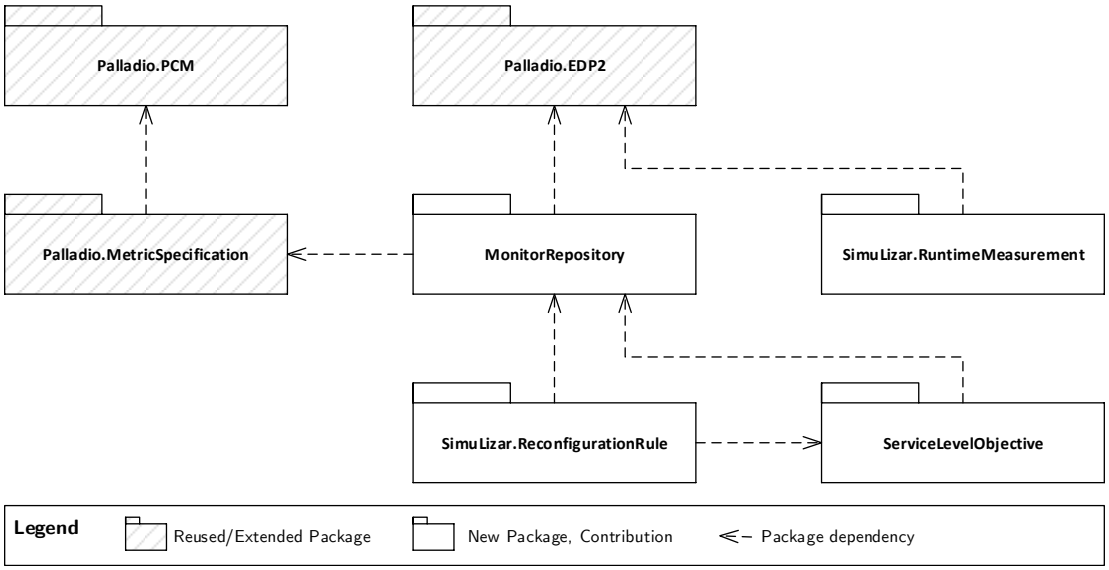
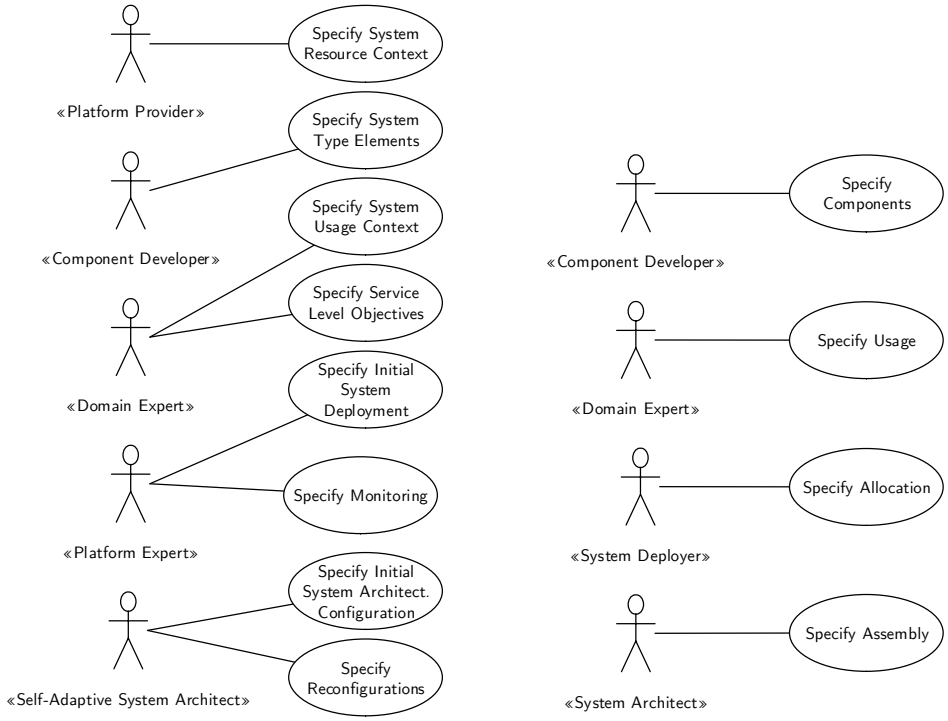


Figure 4.4.: Package diagram of SimuLizar's modeling packages.

be defined for type-level elements, such as components, but also for run-time elements such as concrete system configurations.

Figure 4.4 shows a package diagram illustrating the relationships between the modeling packages in SimuLizar, Palladio/PCM, and DML. Again, the white elements in the diagram are contributions of this work, the crosshatched gray elements are reused from Palladio/PCM or DML. The diagram shows that concepts of Palladio's persistency model EDP2 are used within SimuLizar's MonitorRepository and it's internal run-time measurement model (SimuLizar.Run-timeMeasurement). SimuLizar's reconfiguration view type (SimuLizar.ReconfiguationRule) and ServiceLevelObjective view type build upon concepts of the MonitorRepository.

In summary, the viewpoints in our self-adaptive system performance modeling approach build on top of Palladio's modeling approach and the PCM. We added the self-adaptation viewpoint for modeling the specifics of self-adaptive systems. We reuse view types from PCM and DML in the context of our system type viewpoint and run-time viewpoint. In the following paragraphs, we will discuss the various roles that use these view types to model performance models of self-adaptive systems in order to assess these system's scalability and elasticity.



(a) Modeling roles in SimuLizar. (b) Modeling roles in Palladio. [BKR09]

Figure 4.5.: Use case diagrams for comparison of modeling roles.

4.4.2. Modeling Roles

Figure 4.5 shows the roles that contribute artifacts to the performance models in SimuLizar and Palladio. The left side shows a use case diagram with the roles in SimuLizar that contribute to a self-adaptive system performance model. The right side shows a use case diagram with the roles in Palladio that contribute to a non-adaptive system performance model. The comparison of both sides shows that (1) there are more roles in SimuLizar and the role names in SimuLizar differ from the roles in Palladio, and (2) the use cases in SimuLizar are extended in comparison to Palladio's use cases.

In SimuLizar, we defined a new role, the platform provider. The role of a platform provider is to specify the resource context, i. e., the different element types that, for example, a cloud platform provider offers to its customers. Resource context element types can be (virtual) resources or platform interfaces and components. For example, a PaaS provider will provide interface and component specifications for her provided platform.

An IaaS provider will provide specifications of the virtual resources, such as virtual machines, she provides.

Both, in SimuLizar as well as in Palladio, the component developer's role is to provide reusable specifications of system type elements, like interface specifications and component specifications. In SimuLizar, however, we renamed the use case of the component developer to specify system type elements in order to have a consistent naming with the viewpoints, as presented above.

The domain expert exists in both modeling role concepts as well. However, whereas the domain expert's task is only to specify usage in Palladio, in SimuLizar the domain expert's task is also to specify service level objectives. Service level objectives are highly domain-dependent, e. g., response times which might be acceptable for a news site on the web might not be acceptable for games on a mobile device. Hence, the domain expert has to specify service level objectives. Furthermore, the use case specify usage in Palladio is extended and renamed to specify system usage context in SimuLizar. This use case now additionally includes the specification of time-dependent variation of the system usage context.

The platform expert role in SimuLizar is a specialization of the system deployer role in Palladio. A platform expert has deep knowledge about an execution platform, like a concrete server hardware, an operating system, an IaaS platform, or a PaaS platform. Thus, the platform expert can specify an initial deployment as well as specify monitoring for a self-adaptive system.

Finally, the self-adaptive system architect role in SimuLizar is a specialization of the system architect role in Palladio. A self-adaptive system architect reuses system type elements to specify an initial system architecture configuration, i. e., specifies an assembly of system type element instances to a concrete system. Furthermore, the self-adaptive system architect's role is to specify reconfigurations, i. e., specify how the self-adaptive system gets from one system configuration to another system architecture configuration. In Palladio, the system architect only specifies a single assembly, since systems are considered to be non-adaptive in Palladio.

In conclusion, the roles in SimuLizar refine roles that can be found in Palladio as well. With the platform provider, we added a new role to take the new paradigm of resource leasing in cloud computing into account. We also added new use cases for existing roles, like the specification of monitoring for the platform expert or the specification of service level objectives for the domain expert. These additions reflect new activities for the mentioned roles that come with the paradigm of self-adaptation.

4.4.3. Modeling Process

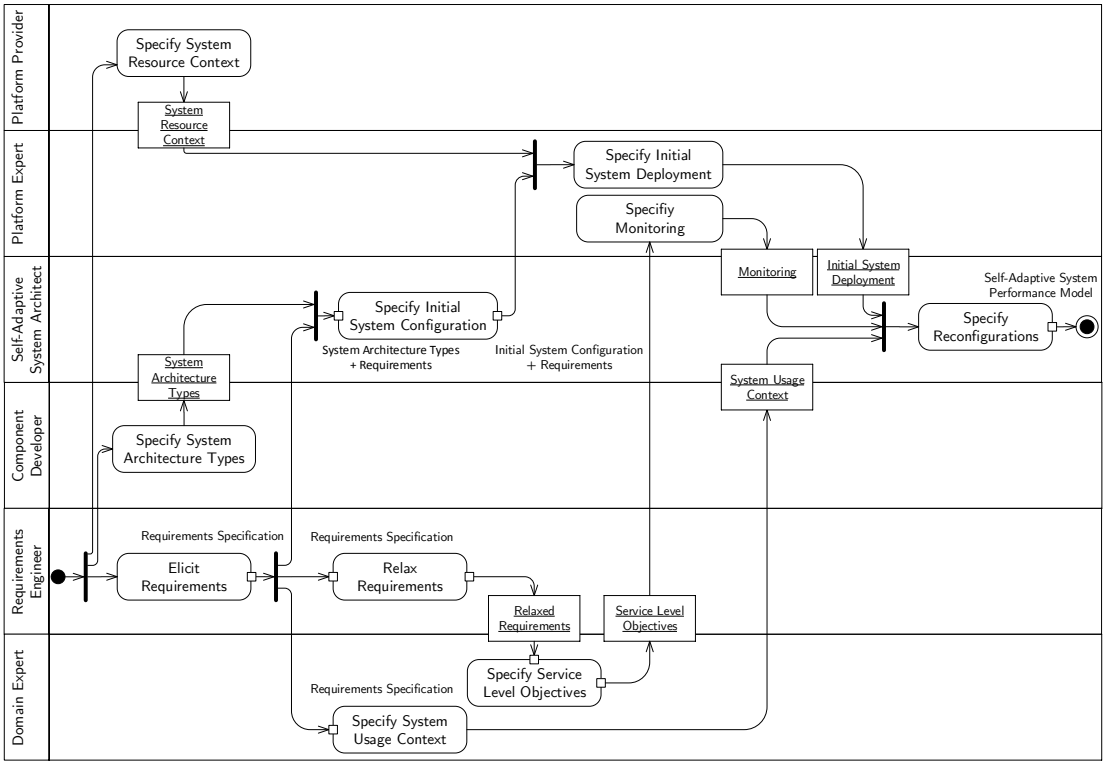


Figure 4.6.: UML activity diagram of the modeling process in SimuLizar.

Figure 4.6 shows SimuLizar's modeling process with the activities for each of the above defined roles. The modeling process starts with concurrent activities of the requirements engineer, component developer, and platform provider. The component developer and platform provider can individually provide specifications for the provided system element types, i. e., interfaces and components, or the provided system resource context, i. e.,

virtual resources. Independent from these activities, the requirements engineer can elicit requirements for the self-adaptive system. Note that the requirements engineer was not introduced above, since this role is part of the requirements engineering phase and not the design phase of the software lifecycle. However, the requirements engineer builds the interface to the design phase since he contributes the input artifacts, i. e., requirements specification, for the design phase.

The output of the elicit requirements activity, i. e., a requirements specification, is used by (1) the self-adaptive system architect to specify an initial system architecture configuration, by (2) the domain expert to specify the system usage context, and by (3) the requirements engineer itself to relax the requirements and thus identify possible trade-offs and monitors. Requirements are relaxed by reformulating requirements as RELAX requirements, as introduced in Section 2.2.1. All of the three activities can happen independent from each other in parallel. However, the output of the component developer's specify system element types activity is required by the self-adaptive system architect in the specify initial system architecture configuration activity. This is reflected by the data flow in Figure 4.6.

After the requirements engineer finished the relax requirements activity, the domain expert's next activity is to specify service level objectives. Independent from this activity, the platform expert can specify an initial system deployment. For this activity, she will need the output from the platform provider's specify system resource context activity. Next, the platform expert will specify monitoring using the output of the domain expert's specify service level objectives activity.

Finally, when all other activities are completed, the self-adaptive system architect can specify reconfigurations. After this last activity, the modeling process is finished and a performance model of a self-adaptive system is completed.

In the following subsections, we describe and illustrate each of the above-mentioned view types, the involved roles, and their activities in more detail. For each view type we (1) describe the purpose of the view type, (2) illustrate the application of the view type on our Znn.com example, (3) explain the view type specification, i. e., its meta model, and (4) discuss the view type's contribution to the fulfillment of our requirements concerned with the scope of the viewpoints.

4.5. System Type Viewpoint

In the system type viewpoint, elements of the self-adaptive system are modeled on type level, i. e., elements are modeled that are not specific to a single system but whose instances can occur within one or many systems at run-time. The system type viewpoint includes two view types and involves two roles. The involved roles are the component developer and the platform provider. The component developer provides specifications of the system architecture types, i. e., interfaces and components, she provides. A specification of the system resource context, i. e., the specification of the platform and its resources, is provided by the platform provider. In the following subsections, we detail view types in the system type viewpoint and provide examples for their artifacts.

4.5.1. System Architecture Types

In SimuLizar, component developers model elements at design-time in the system architecture types view. All system architecture types are on type level and can be instantiated to configure a concrete system architecture configuration. System architecture types are interfaces and component types including a parametric resource-demand specification.

Figure 4.7 shows the system architecture types view for our Znn.com example system. Remember that the Znn.com system has a three-tier architecture with a load balancer. Hence, the view shows the three tiers: INewsFrontend defines the presentation tier, INewsService defines the application tier, and IDatabase defines the data tier. As a refinement of the high-level architecture presented in Section 1.1, the ILoadBalancer interface defines the load balancing interface between the presentation tier and the application tier. These four interfaces are shown in the upper part of the model view depicted in Figure 4.7. Below the interface definitions, five component types are shown. The component types MultimediaNews and TextNews provide the same application logic interface INewsService for the application tier. The component types WebFrontend, LoadBalancer, and Database provide individual interfaces, which can be associated with the presentation tier, load-balancer, and data tier, respectively.

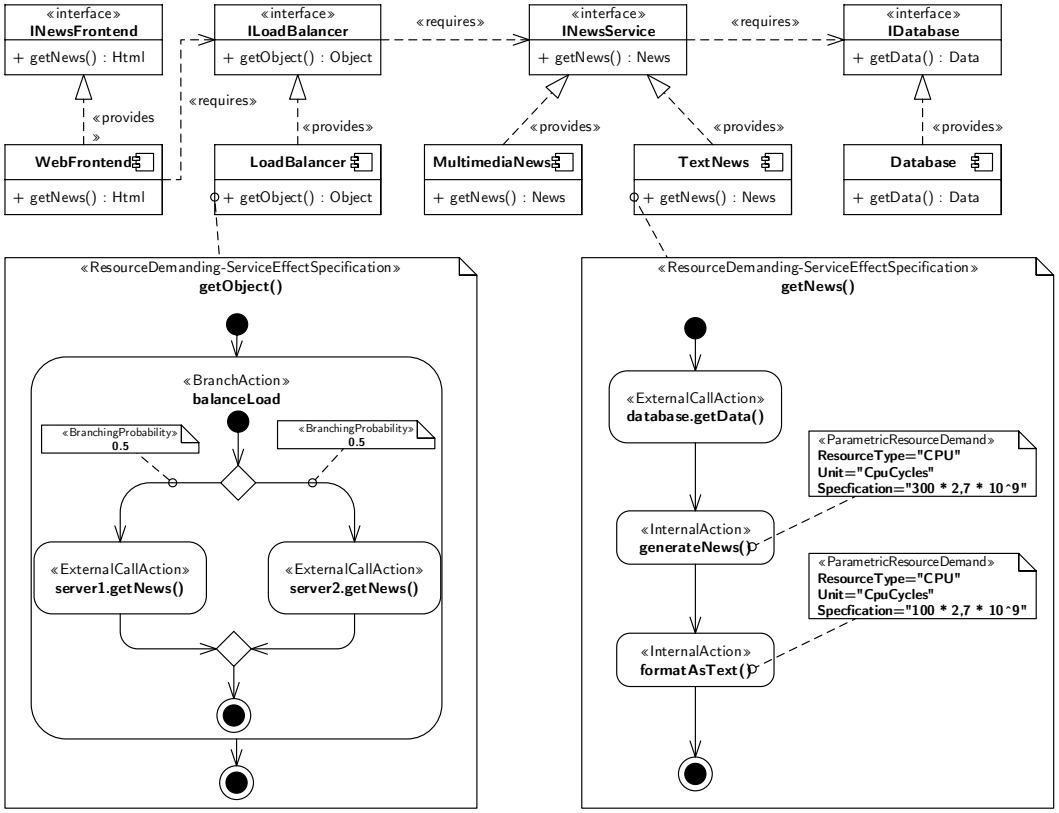


Figure 4.7.: System architecture types for the Znn.com system.

The lower part of the model shows two behavior specifications. The left behavior specification shows the behavior of the LoadBalancer's getObject method, as specified in the ILoadBalancer interface. The getObject method defines a probabilistic «BranchAction» in which one of the two branches is being executed, each with a probability of 50%. In the first branch, the getNews method of the first server is called, in the second branch the getNews method of the second server is called. In this way, the workload is statistically equally distributed between the two servers because each request is delegated with the same probability either to the first server or to the second server. The right behavior specification shows the behavior of the method getNews of the TextNews component. The first action in this behavior specification is an ExternalCallAction, i. e., an interface method call. In this example, the action calls the getData method of the IDatabase interface. The interface method getData is realized by the Database component, which implements the IDatabase interface. After this action, two InternalAction follow sequentially. Both actions include a ParametricResourceDemand specification. The ParametricResourceDemand specifies a parametric demand of an action for a certain resource type, e. g., CPU time or hard disk access time. The actual resource demand depends on the instance of the component with its actual resource demand parameters. In this example, the first action has a resource demand of $300 \times 2,7 \times 10^9$ CPU time units and the second action a resource demand of $100 \times 2,7 \times 10^9$ CPU time units.

Figure 4.8 shows an excerpt of the PCM meta model. In this excerpt, the classes Interface and RepositoryComponent and their respective subclasses OperationInterface, BasicComponent, and CompositeComponent, are defined.

A RepositoryComponent is a NamedElement, i. e., the software architect can assign a human-readable name to it, via the name attribute. RepositoryComponent is abstract and has two concrete subclasses, i. e., CompositeComponent and BasicComponent. A CompositeComponent composes multiple RepositoryComponents (not shown in the excerpt). BasicComponent can include arbitrary many ServiceEffectSpecifications. Software architects specify component behavior with these ServiceEffectSpecifications for each Signature. The Signature class has a subclass OperationSignature, which represents the signature of a method like e. g., getNews(): Html in our Znn.com example. The getNews(): Html method signature defines, the method getNews has no input parameters, i. e., no parameters are listed in the brackets, and the output parameter of

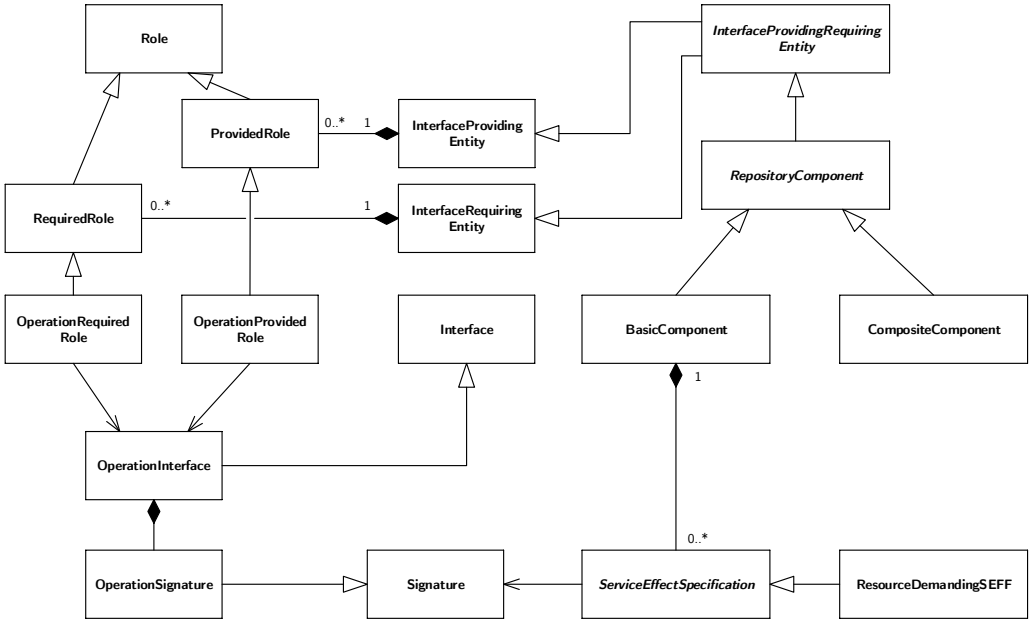


Figure 4.8.: PCM repository meta model excerpt. [RBB+11]

type Html. In general, the characteristics of the parameters of a method can be used within its behavior specification. The behavior of a component type is specified via the ServiceEffectSpecification.

The concrete subclass of the Interface class, OperationInterface, can include multiple OperationSignatures. An OperationSignature is a subclass of Signature, which is referenced by a ServiceEffectSpecification.

ResourceDemandingSEFF (resource demanding service effect specification) is a subclass of the behavior specifications, i. e., ServiceEffectSpecification, which additionally includes the specification of resource demands. The resource demands are used in the analysis in order to predict scalability and elasticity.

Please refer to [BKR09] for a more comprehensive description of the abstract and concrete syntax of the PCM repository model.

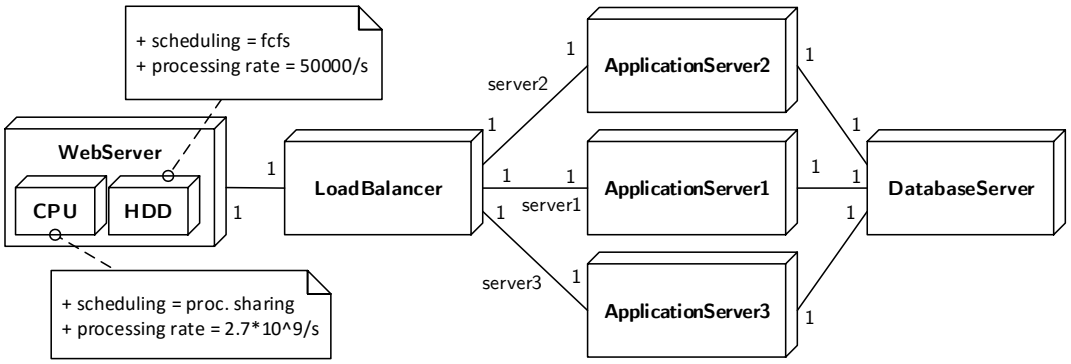


Figure 4.9.: System resource context for the Znn.com system.

In our modeling requirements, we stated that it shall be possible for software architects to specify system architecture types (MR5), specifically structural aspects (MR5.1) like interfaces and components and behavioral aspects (MR5.2) like performance-relevant behavior. All of these requirements are already fulfilled by the PCM's repository model, as outlined above.

4.5.2. System Resource Context

Platform providers, e. g., IaaS provider or PaaS provider, provide a specification of their platform using the system resource view type. The specification of the platform, in SimuLizar also referred to as the system resource context, includes the specification of active resources, e. g., virtual machines or CPUs in an IaaS resource context, and passive resources, e. g., thread pools or connection pools in a PaaS resource context.

Figure 4.9 illustrates the IaaS system resource context for our Znn.com example system. The system resource context consists of a node WebServer, a LoadBalancer node, three ApplicationServer nodes, and a DatabaseServer node. The nodes are connected according to the three-tier architecture, i. e., the presentation tier (WebServer) is connected to application tier (ApplicationServers) via a load balancer (LoadBalancer) and the application tier is connected to the data tier (DatabaseServer). Exemplary, we added a CPU node and a hard disk node to the (WebServer) and specified their capacity and scheduling policy. Note that a platform expert also needs to specify the CPU nodes

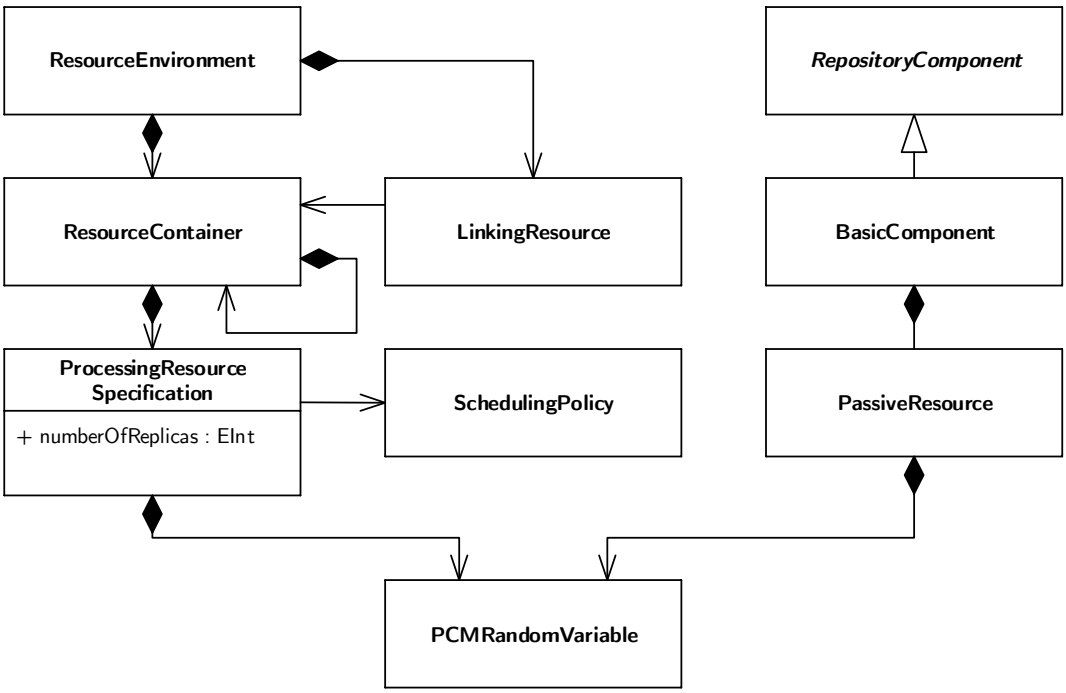


Figure 4.10.: PCM resource environment meta model excerpt. [RBB+11]

and other resources for all elements of the platform. In the example, the (WebServer) has a CPU with a processing rate of 2.7×10^9 and the CPU's scheduling policy is processor sharing. That means, that the WebServer can process 2.7×10^9 CPU operations in one time unit and concurrent operations are scheduled according to a processor sharing policy. Analogously, the hard disk drive has a processing rate of $50,000$ and its scheduling policy is first-come-first-serve. Hence, the WebServer can process $50,000$ hard disk operations, i. e., write or read, within one time unit. Concurrent hard disk operations are scheduled according to the first-come-first serve policy. This specification of the resource capacity is used later in the scalability and elasticity analysis together with the resource demands of system architecture types. Furthermore, nodes that are defined in the resource context view can be replicated via reconfigurations and are thus considered as elements on type level.

The meta model in Figure 4.10 is an excerpt of the PCM meta model showing the ResourceEnvironment view type. We use the ResourceEnvironment view type in SimuLizar

to model active resources in the system resource context. In PCM, a ResourceEnvironment contains arbitrary many ResourceContainers and LinkingResources. A ResourceContainer represents a physical or virtual resource, like a (virtual) server. A LinkingResource represents a physical or virtual link between ResourceContainers. This link can be any type of network connection, wired or wireless and physical or virtual. ResourceContainers contain so-called active resources. Active resources are represented by ProcessingResourceSpecifications in PCM and SimuLizar. The ProcessingResourceSpecification specifies the capacity of the active resource in terms of processed tasks per time unit, i. e., a contained PCMRandomVariable, like introduced in Section 2.1, and the number of identical replicas of the active resource, i. e., numberOfReplicas. For example, if a platform provider wants to specify a server with a $2.4GHz$ dual core CPU, she has to model one ResourceContainer "Server" for the server that contains one ProcessingResourceSpecification for the CPU. The numberOfReplicas attribute of the CPU ProcessingResourceSpecification has to be 2 and the processing rate can be modeled with the PCMRandomVariable "$2.4 * 10E9$". Additional to the capacity specification, platform providers have to specify the SchedulingPolicy for a ProcessingResourceSpecification, i. e., the scheduling algorithm that is used to schedule concurrent tasks for one active resource. PCM provides standard scheduling policies like processor sharing, first-come-first-serve, or delay.

Passive resources are modeled in the PCM repository as elements of a component. On the right side of Figure 4.10, the meta model excerpt of PCM is shown that specifies PasssiveResources. A BasicComponent is a subclass of RespositoryComponent, as explained in the previous subsection. BasicComponents can include arbitrary many PassiveResources. The capacity of a PassiveResource is specified via a PCMRandomVariable. If, for example, a platform expert wants to specify a PaaS database component with a limited connection pool of 50 connections, she needs to model one BasicComponent "Database" that contains a PassiveResource "ConnectionPool" with a PCMRandomVariable "50".

Requirement MR6 states that a suitable modeling approach requires means to specify the resource context of a self-adaptive system. Furthermore, software engineers shall be able to specify active resources (MR6.1), passive resources (MR6.2), and the capacity of resources (MR6.3).

The system resource context view type presented above completely fulfills MR6 and its subordinated requirements. We reuse the PCM resource environment view type in SimuLizar to model active resources of the system resource context. Passive resources can be modeled using the PCM repository view. The PCM resource environment view type already allows the specification of active resources, i. e., MR6.1. The PCM repository view allows the specification of passive resources as required by MR6.2. Finally, the specification of resource capacity, i. e., MR6.3, is possible with PCMRandomVariables for active resources as well as passive resources.

4.6. Run-Time Viewpoint

In the run-time viewpoint, elements of a concrete self-adaptive system on instance level are specified. The run-time viewpoint includes three view types and involves three roles. The involved roles are the domain expert, the platform expert, and the self-adaptive system architect. The domain expert specifies the system usage context, i. e., different workload scenarios according to the requirements and her knowledge about the domain. An initial system architecture configuration is specified by the self-adaptive system architect according to the requirements and using the system architecture types. Finally, the platform expert specifies an initial system deployment for the initial system architecture configuration. In the following subsections, we detail the view types in the run-time viewpoint and provide examples for their artifacts.

4.6.1. Initial System Architecture Configuration

In the initial system architecture configuration, system architecture types will be instantiated and their interfaces connected by a self-adaptive system architect. A valid system configuration is a composed component with one provided interface, i. e., the interface that provides the system level entry methods, but no required interfaces. The provided interface of a system configuration must be delegated to an interface inside the system configuration, since the system configuration has no own behavior.

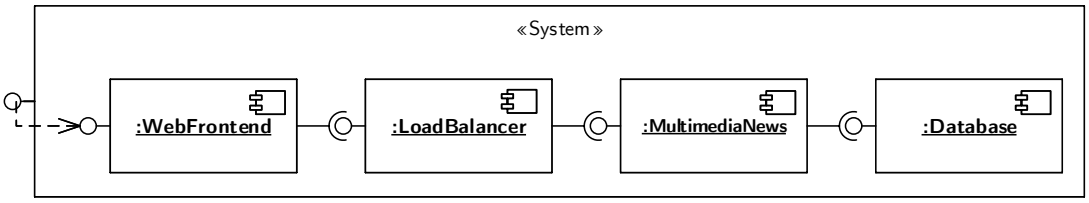


Figure 4.11.: Initial system architecture configuration for the Znn.com system.

The initial architecture configuration can be an arbitrary valid architecture configuration, which will be instantiated at the beginning of the execution of the self-adaptive system. Other architecture configurations may be reached by the specified reconfigurations.

Figure 4.11 shows the initial system architecture configuration for our Znn.com example. It contains four component instances, one from each of the component types WebFrontend, LoadBalancer, MultimediaNews, and Database. The system interface is delegated to the WebFrontend component.

The initial system architecture configuration is modeled using a PCM system model. Figure 4.12 shows the excerpt of the PCM meta model showing the system model classes. System is a subclass off ComposedProvidingRequiringEntity, which is a subclass of ComposedStructure. That means, that a system is a composition of entities, which may provide and require additional interfaces, i. e., component type with their respective interfaces. A ComposedStructure has arbitrary many AssemblyContexts. An AssemblyContext refers to a RepositoryComponent. Therefore, an AssemblyContext represents a concrete instance of a component type within a system.

In our self-adaptation viewpoint requirements, we stated that our viewpoint shall contain a view to configure an initial system architecture configuration (MR7). Furthermore, this initial system architecture configuration shall enable to compose system type level elements to concrete instances (MR7.1) and specify an initial deployment these instances to hardware nodes (MR7.2). We illustrated with our Znn.com example, that we can fulfill these requirements by reusing the PCM system view type.

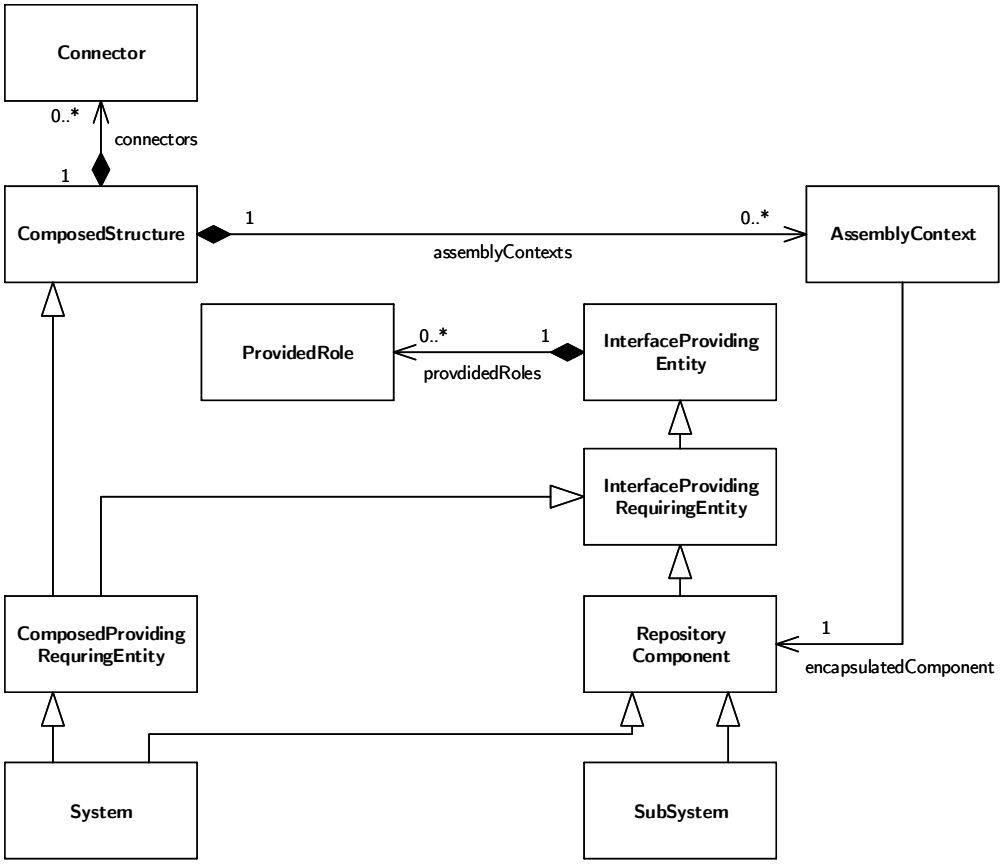


Figure 4.12.: PCM System meta model. [RBB+11]

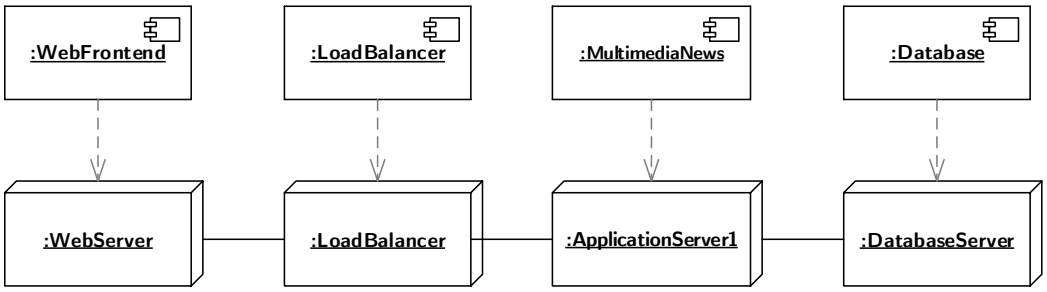


Figure 4.13.: Initial deployment for the Znn.com system.

4.6.2. Initial System Deployment

In the initial system deployment, the deployment of system architecture instances to hardware nodes is specified. Each instance of a system architecture type element that is used in the initial system architecture configuration has to be allocated to exactly one hardware node by the system architect. In a valid system deployment, multiple instances of the same system architecture type element can be allocated to different hardware nodes. However, each instance has to be allocated individually. Not every hardware node needs to be used in the (initial) system deployment. Other system deployments can be reached via reconfigurations. A valid system deployment assigns each instance of a system architecture type element to a hardware node. Thus, hardware nodes not used in the initial system deployment can be used within other system deployments that are reached via reconfigurations.

Figure 4.13 shows the initial system deployment for our Znn.com example. Each component instance is initially allocated on an individual hardware node. In the presented initial system architecture configuration and this initial system deployment, the LoadBalancer component forwards requests to the only MultimediaNews component, which is deployed on the ApplicationServer node. Thus, the performance of each request to the application tier is limited by the ApplicationServer1 node. However, if for example, further MultimediaNews components are instantiated, these component instances can be deployed on the other application server nodes ApplicationServer2 and ApplicationServer3.

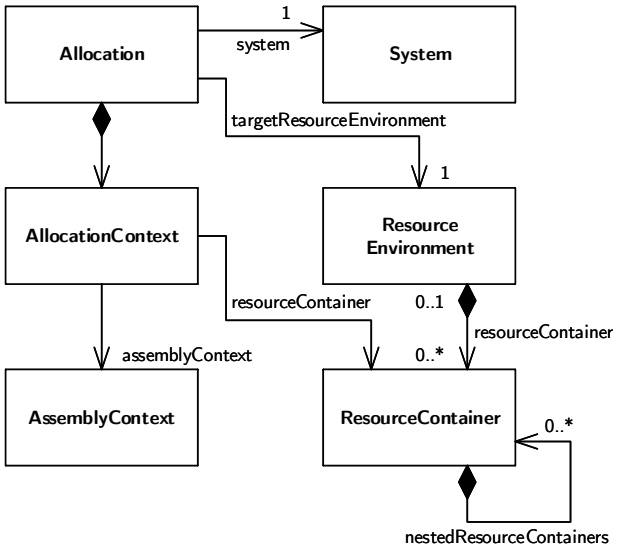


Figure 4.14.: PCM Allocation meta model. [RBB+11]

The PCM allocation meta model is shown in Figure 4.12. We use the PCM allocation view type in SimuLizar to model the initial system deployment. An Allocation refers to a System and ResourceEnvironment and consists of arbitrary many AllocationContexts. An AssemblyContext describes the mapping, or allocation, of AssemblyContexts to ResourceContainers. Hence, each AllocationContext refers to a AssemblyContext, i. e., a concrete instance of a component type within a system, and ResourceContainer from the Allocation's ResourceEnvironment.

In our self-adaptation viewpoint requirements, we stated that our viewpoint shall contain a view to configure an initial system architecture configuration. Furthermore, this initial system architecture configuration shall enable to compose system type level elements to concrete instances (MR7.1) and specify an initial deployment of these instances to hardware nodes (MR7.2). We illustrated with our Znn.com example, that we can fulfill these requirements by reusing the PCM allocation view type.

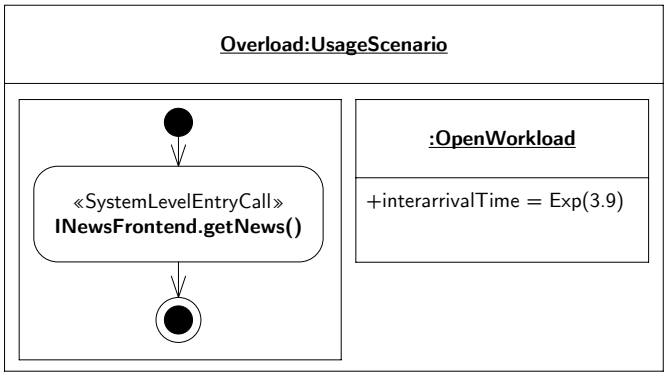


Figure 4.15.: Usage context with one usage scenario for the Znn.com system.

4.6.3. System Usage Context

A domain expert can model typical workload scenarios and the time-dependent variation of the workload in these scenarios in the system usage context view. The scenarios are used for the scalability and elasticity analyses later on. A typical workload scenario consists of the specification of the work and load. The work is specified as a user action flow, i. e., an order of system method calls, called by a certain user type, and a characterization of the respective arguments (actual parameters). The load is specified as a frequency of the method called by a user.

By specifying specific scenarios for various user action flows and situations, software architects can assess the scalability and elasticity of their self-adaptive system designs individually in each of these scenarios, e. g., at peak load or when workload from routine maintenance (other method calls) mix into the standard workload. Furthermore, domain experts can specify time-dependent variations of the typical workload scenarios, e. g., seasonal and periodical workload patterns, random workload bursts, and long-term workload trends.

Figure 4.15 shows one usage scenario of the system usage context for our Znn.com system. The illustrated system usage context specifies a peak-load scenario with a simple work and load specification. The work is defined by the scenario behavior on the left side of the illustration. The scenario behavior consists of a start action, a SystemLevelEntryCall of the getNews method, which is provided to the users by the

Znn.com system, and a stop action. The right side of the system usage context shows the load specification. In this example, the scenario is an OpenWorkload, i. e., the user request rate is independent from previous user requests. The request rate, in this example, is exponentially distributed with $\lambda = 3.9$, i. e., the mean interarrival time of user requests is $1/\lambda \approx 0.25$ time units.

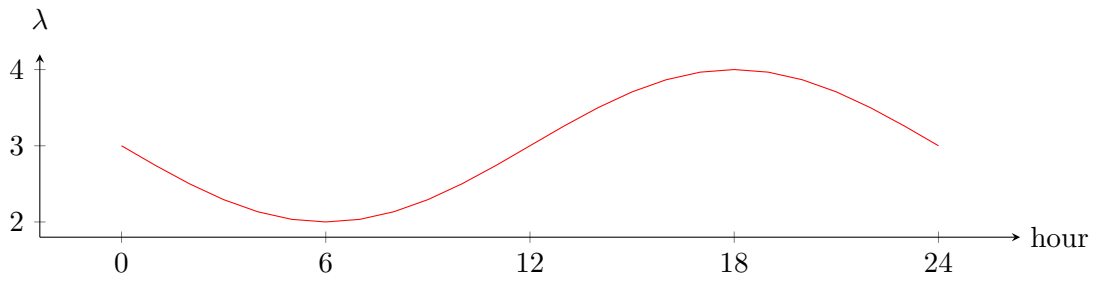


Figure 4.16.: Time-dependent variation of the load in the Znn.com example.

Figure 4.16 shows a simple time-dependent variation of the system usage context for the Znn.com system. The graph shows a periodic increase and decrease of the load for the usage scenario illustrated in Figure 4.15. A periodic variation of the system usage context like illustrated in the figure can be specified with the Seasonal element. Additional to this periodic variation, other variations can be specified with different Function elements, e. g., for long-term workload trends or random workload bursts.

Usage scenarios can be specified with PCM's UsageModel, as illustrated in the meta model excerpt in Figure 4.17. A UsageModel consists of arbitrary many Workloads. Each UsageScenario consists of a ScenarioBehavior that characterizes the work and a Workload that characterizes the load of the scenario. A ScenarioBehavior consists of arbitrary many subclasses of AbstractUserActions, i. e., Start, Stop, Loop, Branch, Delay, and SystemLevelEntryCall. Workload is abstract, hence a UsageScenario must contain one of Workload's subclasses, OpenWorkload or ClosedWorkload. In case of an OpenWorkload, the load is further specified with a interarrivalTime attribute of the type PCMRandomVariable. In case of an ClosedWorkload, the attribute thinkTime, also of type PCMRandomVariable, specifies the load.

Time-dependent variations of the workload in usage scenarios can be specified by a UsageEvolution as shown in the meta model in Figure 4.18. A UsageEvolution contains

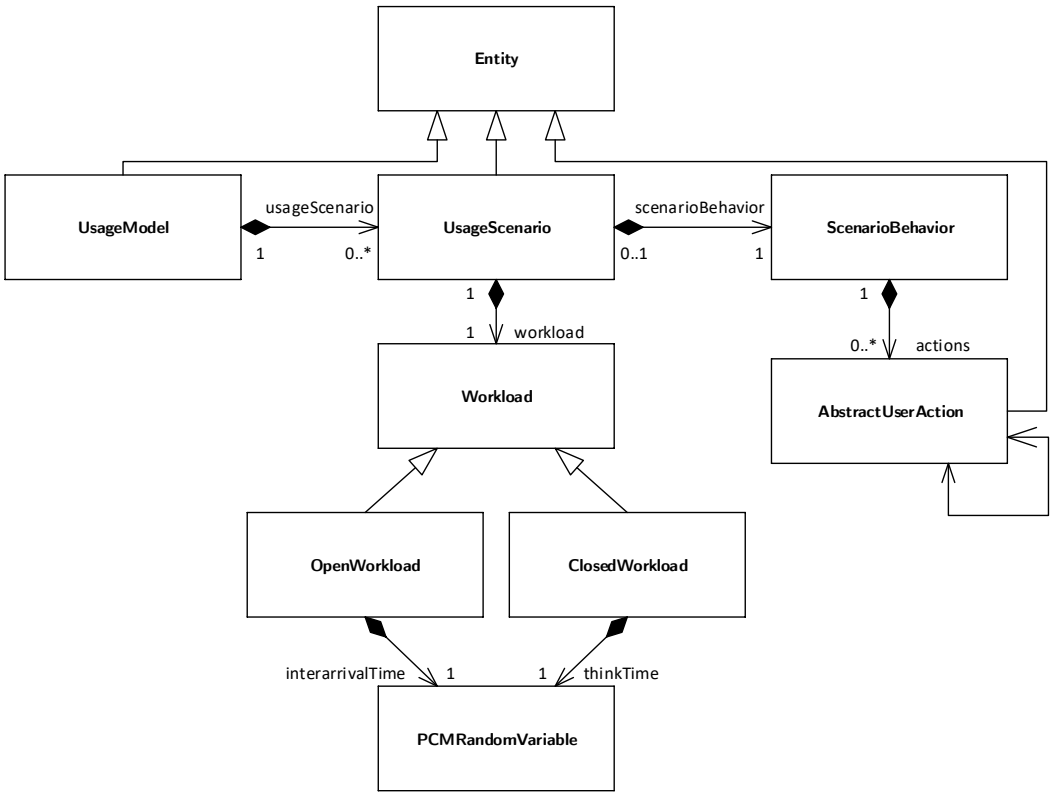


Figure 4.17.: PCM Usage Model meta model [RBB+11]

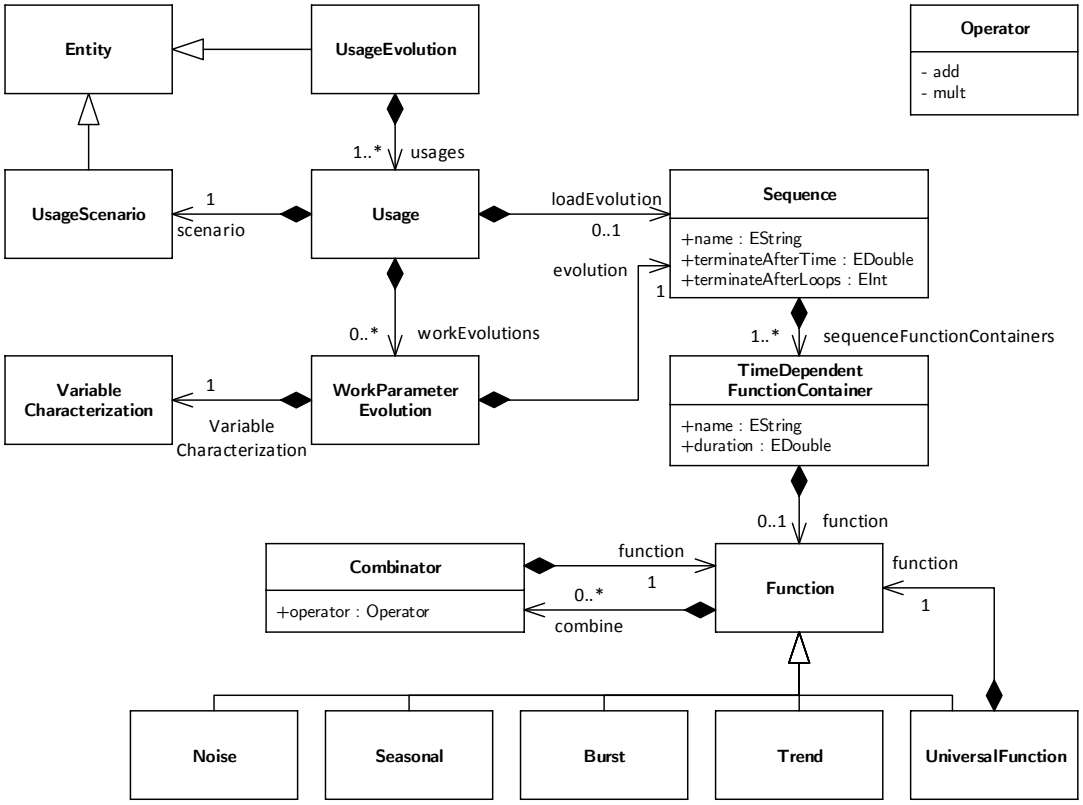


Figure 4.18.: Combination of the Usage Evolution meta model [BSL16] and the DLIM meta model [KHK+17]

arbitrary many Usage elements, which correspond to a UsageScenario, as described above. A Usage element can contain a WorkParameterEvolution to specify the time-dependent variations of a work parameter, for example file sizes that are uploaded to a server. The WorkParameterEvolution then has to refer to the VariableCharacterization that defines the corresponding work parameter. The variation of the work parameter as well as the variation of the load for a Usage are specified in a Sequence element. The Sequence element contains multiple TimeDependentFunctionContainers. Each of these TimeDependentFunctionContainers can contain a concrete Function subclass element, i. e., Noise, Seasonal, Burst, Trend, or UniversalFunction. These functions are used to specify the variation of the work parameters and load for different time periods. The time periods are defined via the duration attribute of the TimeDependentFunction and the attributes terminateAfterTime and terminateAfterLoops of a Sequence.

Our modeling requirements include a requirement for a system usage context view type, i. e., Requirement MR8 and the three sub-requirements Requirement MR8.1, Requirement MR8.2, and Requirement MR8.3. All these requirements are fulfilled by SimuLizar's system usage context view type. The presented view type enables domain experts to model multiple UsageScenarios, i. e., Requirement MR8. With AbstractUserActions the work can be specified, i. e., Requirement MR8.1, and with OpenWorkload and ClosedWorkload the load can be specified, i. e., MR8.2. Additionally, time-dependent variations of the workload scenarios can be modeled with UsageEvolution elements. The workload scenarios specified by a domain expert are used for the assessment of scalability and elasticity of the modeled self-adaptive system within these system usage contexts.

4.7. Self-Adaptation Viewpoint

In the self-adaptation viewpoint, the so far static system architecture is extended with the necessary artifacts for autonomous self-adaptation. The self-adaptation viewpoint includes three view types and involves three roles. The domain experts specifies service level objectives (SLOs) which are the major drivers for self-adaptation, i. e., SLOs define in which borders the self-adaptive system shall operate. When the SLOs are defined,

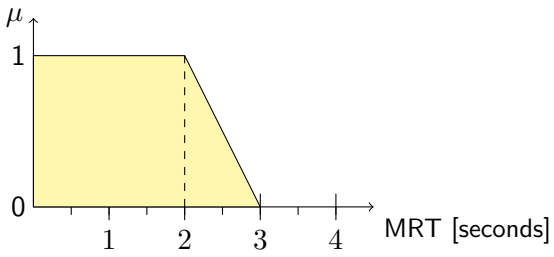
a platform expert specifies monitors that give the self-awareness capabilities for the self-adaptive systems. Finally, a self-adaptive system architect specifies reconfigurations that define when the system needs to reconfigure its architecture in order to achieve the SLOs. In the following subsections, we detail the view types in the self-adaptation viewpoint and provide examples for their artifacts.

4.7.1. Service Level Objectives

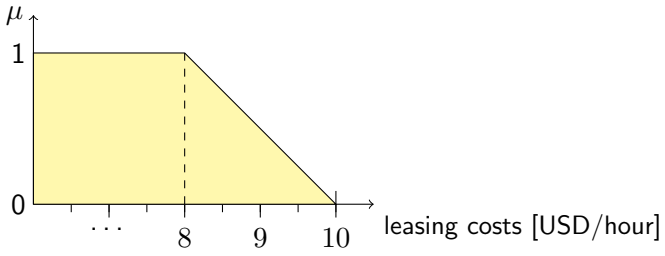
Service level objectives (SLOs) define thresholds for quantifiable quality attributes of software systems, like performance. In self-adaptive systems, SLOs are major drivers for self-adaptation. That is, in case the current system architecture configuration of a self-adaptive system is not able to satisfy an SLO, e. g., the target mean response time of 2.0 seconds cannot be achieved, the system will detect this situation and may execute a reconfiguration to switch to another system architecture configuration. In Section 1.1, we defined the five requirements (R1 to R5) for our Znn.com example, which can be refined to service level objectives as well.

Figure 4.19 illustrates the service level objectives SLO_{MRT} and SLO_{CST} that are derived from the requirements of our Znn.com example. We derived SLO_{MRT} from R1, i. e., the requirement that the mean response time shall be less than 2.0 seconds. We relaxed this requirement with our service level objective SLO_{MRT} such that 2.0 seconds is the upper soft threshold and 3.0 seconds is the upper hard threshold. That is, the SLO_{MRT} is completely achieved if the mean response time is below 2.0 seconds and is not achieved if the response time is above 3.0 seconds. If, however, the response time is between 2.0 seconds and 3.0 seconds, requirement SLO_{MRT} is only achieved to a certain grade. SLO_{CST} is analogously defined for requirement R2, i. e., the requirement that the leasing cost shall not be higher than $USD5.00$ per hour. The interpretation of gradual achievement of requirements is based on our formalization of SLOs that we present in more detail in Chapter 5.

Figure 4.20 shows the meta model of the SLO specification model. The root element of the SLO model is the ServiceLevelObjectiveRepository, which contains all ServiceLevelObjectives. A ServiceLevelObjective consists of at least one Threshold, which is ensured



(a) SLO_{MRT}



(b) SLO_{CST}

Figure 4.19.: Service level objectives for our Znn.com system.

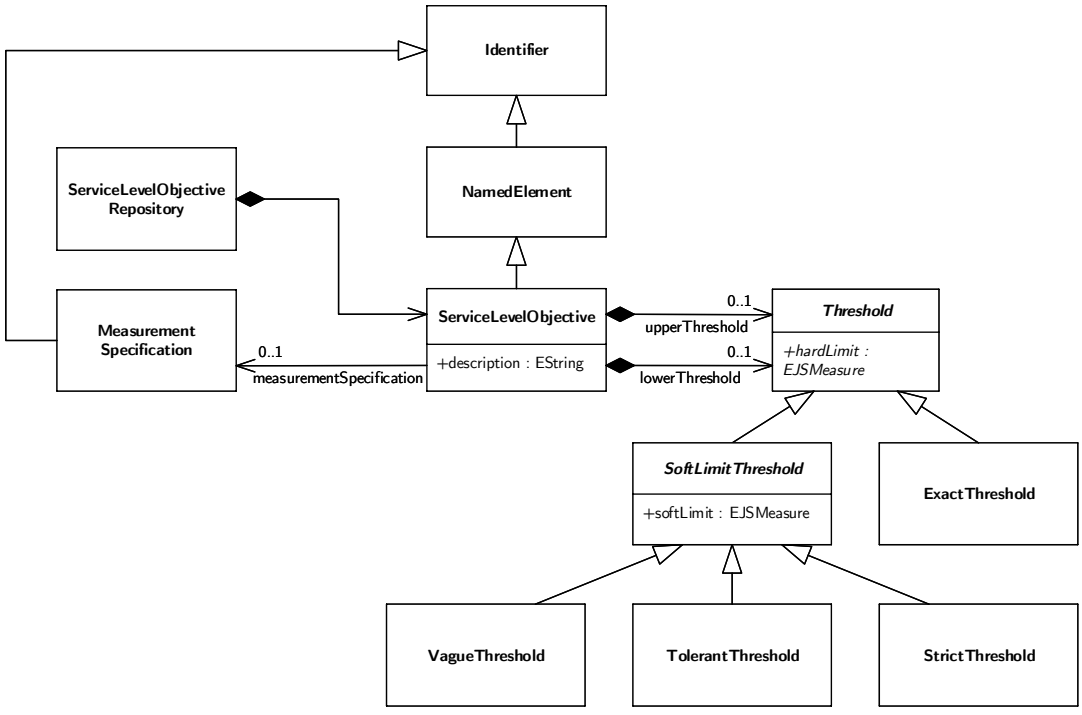


Figure 4.20.: Service level objective meta model.

by the static semantics, specified in in the Object Constraint Language (OCL), of the meta model. Thus, a ServiceLevelObjective can have one upper threshold, one lower threshold, or both. Furthermore, a ServiceLevelObjective refers to a MeasurementSpecification, which specifies to which specific measurements of the self-adaptive system this SLO refers, e. g., the response time SLO in our Znn.com example.

The class Threshold is abstract and has four concrete subclasses, i. e., VagueThreshold, TolerantThreshold, StrictThreshold, and ExactThreshold. The class ExactThreshold only has the hardLimit attribute, inherited from its superclass. The other classes VagueThreshold, TolerantThreshold, and StrictThreshold inherit the attribute softLimit from their superclass SoftLimitThreshold additionally to the hardLimit attribute from the Threshold class. In our Znn.com example, there are only tolerant thresholds with soft limits. For example, the SLO_{MRT} specifies that the mean response time of the Znn.com system shall be less than 2 seconds if possible (soft limit), but not worse than 3 seconds (hard limit).

We require a view type that enables to specify SLOs, as stated in Requirement MR9. Furthermore, software architects shall be able to specify soft limits and hard limits for thresholds, i. e., Requirement MR9.1 and Requirement MR9.2. These requirements are fulfilled by the different threshold classes as illustrated in the Znn.com examples.

4.7.2. Monitor Repository

With the monitor repository view, monitors for a self-adaptive system can be specified by a platform expert. A single monitor consists of two characteristics: (1) a measuring point and (2) a measurement specification. The measuring point specifies the location of the monitor within the system, e. g., a system architecture type like an interface method. The measurement specification defines which metric shall be monitored at the specified measuring point and how the metric values shall be aggregated over time. For example, the metric *response time* can be aggregated as *mean response time* calculated in fixed-size batches.

The monitor repository for our Znn.com example is illustrated in Figure 4.21. We defined two monitors in this monitor repository. The first monitor is the System Response

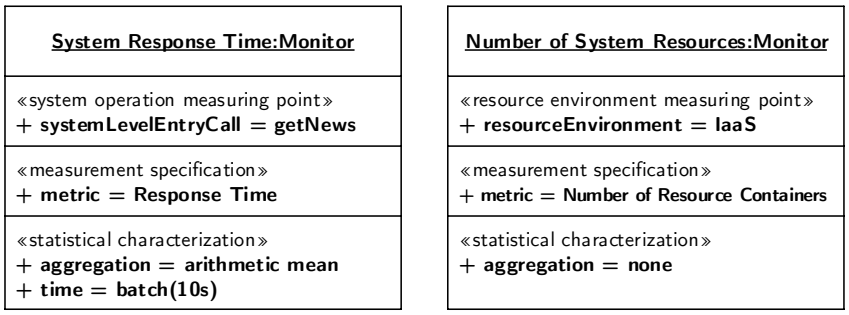


Figure 4.21.: Monitor repository for the Znn.com system.

Time monitor. It monitors the overall mean response time of the Znn.com system, i. e., the time required to process a user request to show the Znn.com website. Hence, the monitor's measuring point is the getNews method, which is System Level Entry Call, i. e., the method a user calls. The measurement specification of this monitor specifies response time as the metric. To calculate the mean response time, the statistical characterization is set to aggregate response times in 10 second batches, i. e., batch(10s), with the arithmetic mean aggregation function. The second monitor in the Znn.com example is the Number of System Resources monitor. With this Monitor, the overall number of resource containers, e. g., virtual machines, is monitored. The measurement specification specifies the monitored metric, which is Number of Resource Containers here. Since we are interested in the absolute number of resources at any time, the aggregation is set to none in the statistical characterization.

Figure 4.21 shows the monitor repository meta model. The root element is the MonitorRepository class. It can contain arbitrary many Monitors. Each Monitor refers to exactly one MeasuringPoint, which specifies the location the measurements shall be collected. The class MeasuringPoint is specified in a separate meta model. For Palladio and SimuLizar, we defined specialized MeasuringPoints, like SystemOperationMeasuringPoint, ResourceEnvironmentMeasuringPoint, and ReconfigurationTimeMeasuringPoint, which allow easy specification of a MeasuringPoint for common monitors in self-adaptive systems. Additionally, a Monitor is composed of one to many MeasurementSpecifications. Each MeasurementSpecification has a name (inherited from NamedElement), a statisticalCharacterization, refers to exactly one MetricDescription, and optionally one TemporalCharacterization. The MetricDescription class is specified in a separate package. In SimuLizar,

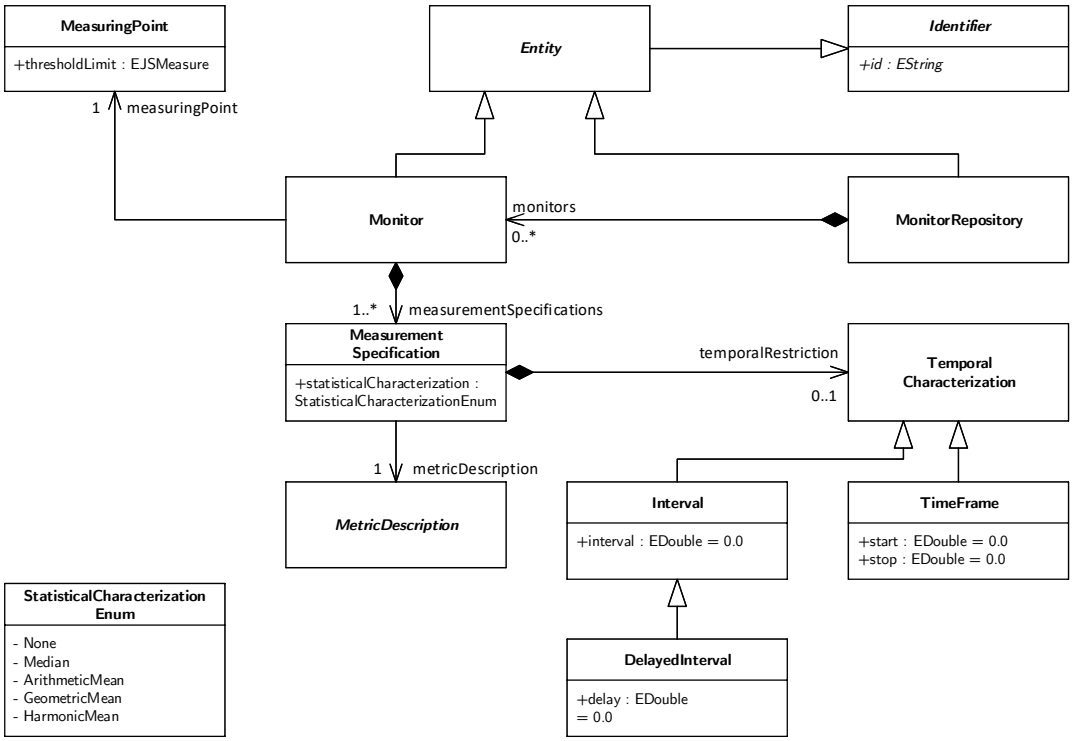


Figure 4.22.: Monitor repository meta model

many predefined MetricDescriptions are available, e. g., response time, reconfiguration time, or generic metrics like unit-less number. The statisticalCharacterization can be either none, median, arithmetic mean, geometric mean, or harmonic mean. The valid values are defined in the enumeration StatisticalCharacterizationEnum. The class TemporalCharacterization is abstract and has three concrete subclasses, which can be used to further specify temporal characterizations of the MeasurementSpecification. These three subclasses are Interval, DelayedInterval, and TimeFrame. Values are aggregated in fixed-size, non-overlapping intervals using Interval as TemporalCharacterization. The interval size is defined with the interval property in time units, i. e., as double value. The DelayedInterval works the same way, but the interval calculation does not start at time 0, but after a specified delay. This is helpful for cases where a warm-up phase of a simulation shall not be taken into account. For example, if we want to consider only mean response times after a warm-up time of 100 time units, we simply use a DelayedInterval as a temporal characterization for the MeasurementSpecification and set the delay property of the DelayedInterval to 100. The response time predictions in the warm-up phase of a system are often not very accurate, since in practice caches and memory are initially filled for the software. However, these effects cannot be simulated with our simulation and thus leading to inaccurate predictions. The third subclass of TemporalCharacterization is TimeFrame. This class can be used for single, fixed-size aggregations, e. g., a single mean response time for the time frame starting after 5 time units and ending after 205 time units. TimeFrame has two properties start and end corresponding to the start and end of the time frame to be aggregated.

In our self-adaptation viewpoint requirements, we have the requirement MR10 consisting out of four subrequirements. From the discussion of the meta model and the provided example, we can see that all these requirements are fulfilled with our Monitor Repository model. First, we can specify the monitor location, as required by MR10.1 using MeasuringPoints. Second, the MeasurementSpecification allows specifying the metric and the aggregation of metric values as required by MR10.2 and MR10.4. Finally, the specification of the frequency for taking monitor probes is specified within TemporalCharacterization of a MeasurementSpecification as required by MR10.3.

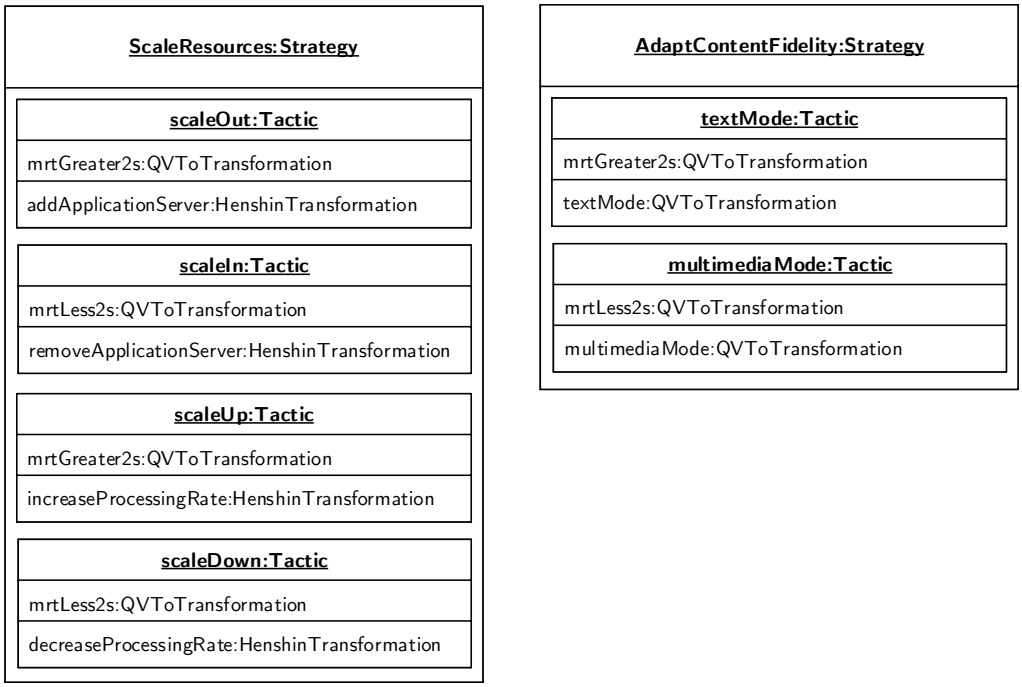


Figure 4.23.: Reconfiguration rule model for the Znn.com system.

4.7.3. Reconfigurations

SimuLizar's reconfiguration view type provides a way to specify self-adaptations in a hierarchical style. Reconfigurations are divided into high-level strategies and mid-level tactics, and low-level actions. For example, self-adaptive system architects can specify multiple strategies to reach the same goal. If the goal is to keep response times low, two alternative strategies can be (a) scaling the system resources, i. e., adding more virtual machines, or (b) reducing the content fidelity, like the text-only mode in our Znn.com example. A tactic is a concrete implementation specification of a strategy, i. e., it aggregates all necessary actions and their preconditions to execute the strategy. In SimuLizar, the preconditions and actions are model-transformations. These model-transformations either check for a certain condition or modify the SimuLizar model with an in-place transformation depending on whether the transformation is a precondition or an action.

Figure 4.23 shows an example reconfiguration view for our Znn.com example. The shown reconfigurations view contains two alternative reconfiguration strategies: (1) scaling resources and (2) adapting content fidelity. The first strategy takes advantage of the cloud computing environment in which the Znn.com system is running in. The second strategy, i. e., adapting the content fidelity, switches the content delivered to customers between text-only and multimedia, e. g., including images and video.

The first strategy, scaling resources, includes four tactics, where the two rules scaleOut and scaleIn are tactics with opposite effects. Analogously, the tactics scaleUp and scaleDown are tactics with opposite effects. scaleOut and scaleIn replicate the application tier of the Znn.com system, i. e., these tactics add/remove instances of the MultimediaNews component to the system configuration and add these to the load balancing behavior of the LoadBalancer component. The tactics scaleUp and scaleDown do not replicate the tiers but increase/decrease the processing rate of the application tier's resource containers, i. e., the ApplicationServers.

Listing 4.1: Precondition "mrtGreater2s" for "scaleOut" Reconfiguration Tactic

```
1  property threshold : Real = 2.0;
2
3  main() {
4
5    assert fatal(run-timeMeasurement.rootObjects()[Run-timeMeasurement
       ]->size() > 0)
6      with log ("No Measurements found!");
7
8    assert error (run-timeMeasurement.rootObjects()[Run-timeMeasurement
       ]->checkCondition() = true)
9      with log ("No reconfiguration required");
10
11 }
12
13 helper Set(Run-timeMeasurement) :: checkCondition() : Boolean {
14   self->forEach(measurement) {
15   log('Measured value is ' + measurement.measuringValue.toString());
16     if (measurement.measuringValue > threshold) {
17       log('Threshold is exceeded');
18       return true;
19     };
20   };
21   return false;
22 }
```

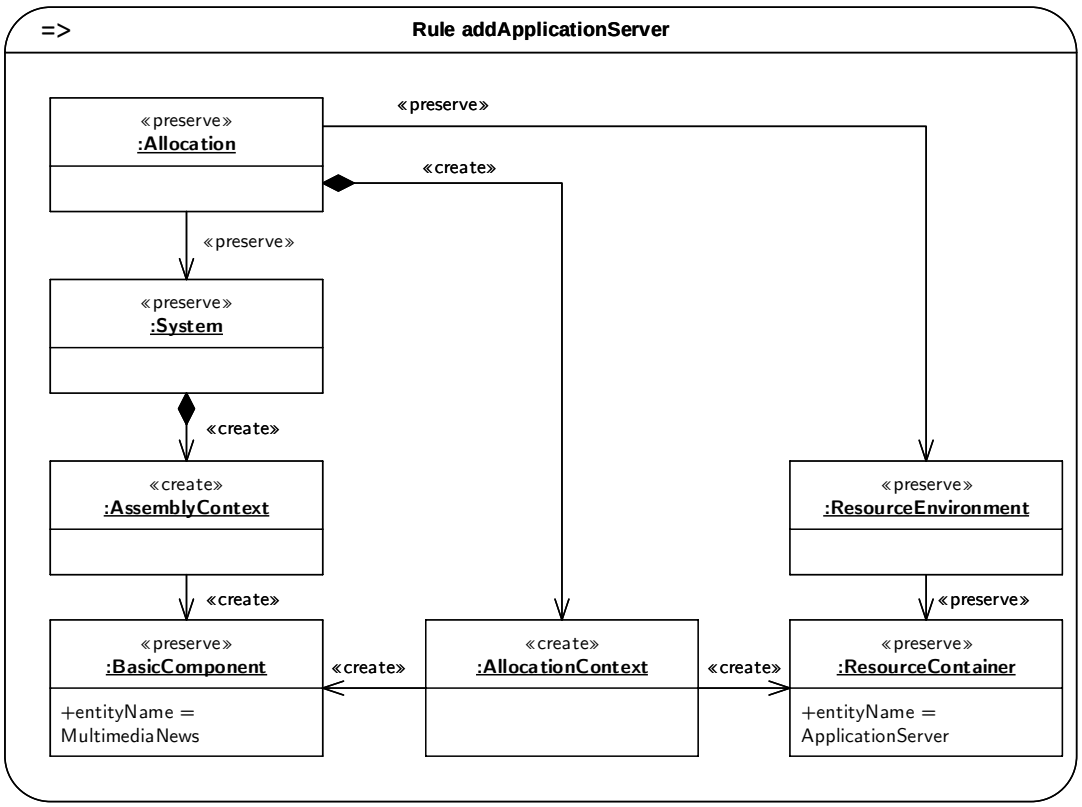


Figure 4.24.: Action element of the “scale out” tactic for the Znn.com system.

Figure 4.24 shows the action element of the scaleOut tactic using a Henshin model transformation. The transformation adds an instance of the MultimediaNews component type to the system configuration and allocates it to an ApplicationServer node. Note, that Henshin model transformations and Henshin diagrams, like the one in Figure 4.24, use the abstract syntax of the models and not a concrete syntax. In Henshin, elements that are only matched without being modified are annotated with the «preserve» stereotype, newly created elements are annotated with the «create» stereotype, and elements to be deleted are annotated with the «delete» stereotype. The AssemblyContext that is created as a child element of the System is an instance of the MultimediaNews component type, i. e., in abstract syntax this is a BasicComponent with entityName "MultimediaNews". For this instance, an AllocationContext is created as well. This AllocationContext describes the allocation of the component instance to a ResourceContainer, here the ApplicationServer.

The precondition for this action is specified with a QVT model transformation and is illustrated in Listing 4.1. Note that the listing shows only the relevant excerpt of the QVT operational reconfiguration precondition. The declaration of `modeltypes`, the definition of the `transformation` body with input and output parameters, and additional checks are left out in the listing. In line 1 the variable `threshold` for the maximum mean response time is declared and initialized to 2.0, i. e., a mean response time threshold of 2.0 seconds. The main function (lines 3 to 11) of the QVT operational rule includes two assertions: An assertion that the run-time measurement model is not empty (lines 5 and 6), and an assertion that the `checkCondition` helper function returns `true`. In case the first assertion fails, the rule will not be further executed. In case the second assertion fails, the check returns an error to signal that the action transformation of the reconfiguration rule must not be executed. The helper function `checkCondition` (lines 14 to 22) returns `true` if there is any measured value over the threshold value in the run-time measurement model, see Chapter 6. If there is no measured value over the threshold value, the helper function will return false and the second assertion will fail.

The second strategy, adapting the content fidelity, includes the two tactics textMode and multimediaMode. The tactic textMode degrades the content fidelity of the news content delivered to customers in favor of performance, i. e., lower response times for news requests. Figure 4.25 shows the action of the textMode tactic implemented as

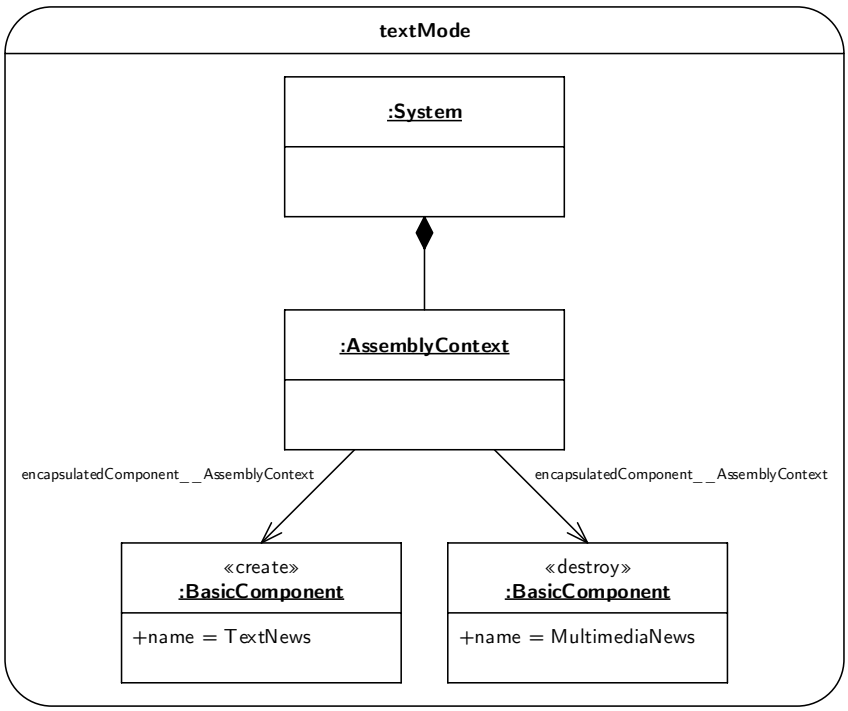


Figure 4.25.: Action element of the “textMode” tactic for the Znn.com system.

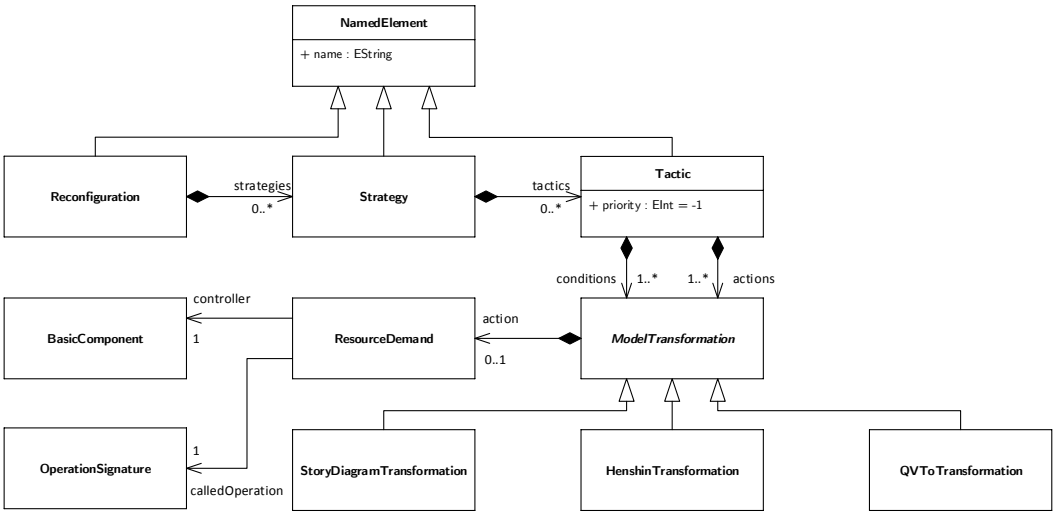


Figure 4.26.: Reconfiguration rule meta model

a Story Diagram model transformation. The transformation exchanges all MultimediaNews component instances for TextNews component instances. Note that the Story Diagram transformation uses the abstract syntax instead of the concrete syntax of the PCM or SimuLizar view types. Hence, component instances of type MultimediaNews are represented by a AssemblyContext with a encapsulatedComponent__AssemblyContext relation to a BasicComponent with entityName "MultimediaNews". The tactic multimediaMode is implemented analogously to the textMode tactic.

Figure 4.26 shows the meta model of the reconfigurations view type. A Reconfiguration has arbitrary many Strategy objects. Each Strategy again, can have arbitrary many Tactics. In our Znn.com example, there are two strategies: (1) resource scaling and (2) content fidelity adaptation. Each tactic has one or more conditions and one or more actions. The type of both, conditions and actions, is ModelTransformation. ModelTransformation is a generic type, which has the three concrete subtypes QvtoTransformation, StoryDiagramTransformation, and HenshinTransformation. Each of these concrete subtypes represents transformation scripts from their respective transformation language QVT operational, Story Diagrams, and Henshin. In that way, it is possible to mix these different transformation languages for conditions and actions within one reconfiguration tactic. A ModelTransformation also contains a ResourceDemand which refers to

a BasicComponent and an OperationSignature. The referenced BasicComponent represents the controller component for the reconfiguration, i. e., the component that executes the reconfiguration. The OperationSignature represents an operation, i. e., specified as ResourceDemandingSEFF, of that component. Hence, the resource demand of a reconfiguration is modeled like a normal behavior in the system architecture type view type.

Our modeling requirements that address the specification of architecture reconfigurations, i. e., Requirement MR11 and its sub-requirements, are fulfilled by the presented reconfigurations view type. As required by Requirement MR11.1 and Requirement MR11.3, architecture reconfigurations and preconditions can be specified with a Tactics, that can include multiple conditions and actions. The resource demand of a reconfiguration action, i. e., Requirement MR11.2, can be specified with a ResourceDemand that refers to an OperationSignature of a BasicComponent. A self-adaptive system architect can hence model architecture reconfigurations including preconditions, reconfiguration actions, and resource demands with SimuLizar. These architecture reconfigurations are used for the prediction of scalability and elasticity properties of the modeled self-adaptive system. The reconfiguration actions also allow an exploration of the reconfiguration space, i. e., the exploration of all reachable architecture configurations. This allows the prediction of scalability. Furthermore, the resource demands of reconfiguration actions allow the prediction of elasticity properties since these resource demands influence how quick a self-adaptive system can adapt in certain usage contexts.

4.8. Evaluation

In the previous sections, we introduced three viewpoints for our self-adaptive system performance modeling approach. The goal of the presented modeling approach is to enable the specification of self-adaptive system architectures for the purpose of analyzing these system's performance properties. In this section, we validate our approach regarding the achievement of this goal.

Böhme and Reussner [BR08] describe three validation levels for prediction methods. A Level I validation is concerned with the validation of the predicted metrics. For this kind of validation, predictions are compared with measurements to validate the

homomorphism of the predictions provided by the prediction method with respect to the reality. A Level II validation is concerned with the applicability of the prediction method. Hence, in this kind of validation it is evaluated whether the target group, i. e., software engineers, can reliably produce the input for the prediction method and meaningfully interpret the output of the prediction method [BR08]. A Level III validation is concerned with the benefits of the prediction method compared to other prediction methods. This is kind of validation requires a controlled experiment [BF08] in which, for example, two (or more) prediction methods are applied to the same software system.

A self-adaptive system performance model, like presented in this chapter, is the input for our scalability and elasticity prediction that we present in the next chapter. We present a Level II validation for our self-adaptive system performance model in this section. A Level I validation for our prediction methods is presented with our prediction methods in Chapter 5. Due to the limited time and resources, a Level III validation could not be conducted in the scope of this thesis.

We applied the goal question metric (GQM) approach [vBCR02] to conduct a case study for the Level II validation of our modeling approach. A computer science Master student conducted the case study using the Znn.com system as the modeling case for the validation. The student applied our modeling approach to model the Znn.com system and collected limitations of the approach.

Table 4.3 formulates our evaluation goal within the case study using the GQM template. Our goal is to *analyze* the presented self-adaptive system performance modeling approach *for the purpose of* evaluating the applicability of the approach *with respect to* our modeling requirements MR1 to MR11. The case study is conducted *from the viewpoint of* software engineers. The *context* of the evaluation is academic, i. e., a computer science Master student performed the evaluation.

Our case study showed that the presented performance modeling approach is applicable for self-adaptive systems but still has some limitations. In the remainder of this section, we describe our evaluation and results in more detail. In Section 4.8.1, we formulate the evaluation questions. We explain the evaluation setup in Section 4.8.2. Subsequently, we present the evaluation results in Section 4.8.3. In Section 4.8.4, we discuss the

Table 4.3.: Evaluation Goal

Analyze	SimuLizar's self-adaptive system performance modeling approach
for the purpose of	evaluating the applicability of the approach
with respect to	our modeling requirements MR1 to MR11
from the viewpoint of	software engineers
in the following context:	A computer science Master student models the Znn.com system for the purpose of assessing the system's scalability and elasticity.

evaluation results with respect to our modeling requirements MR1 to MR11. Finally, we discuss the threats to validity in Section 4.8.5.

4.8.1. Questions

We evaluated the applicability of the presented performance modeling approach with respect to our modeling requirements MR1 to MR11 within a case study for the Znn.com example system. We formulated two evaluation questions for the validation of the *applicability* as well as the identification of *limitations* of our approach. For each evaluation question, we define a metric and a hypothesis.

Table 4.4.: Question 1: Applicability

Q(applicability)	**Is SimuLizar's modeling approach applicable to create a performance model of a self-adaptive system?**
M(model homomorphism)	SimuLizar's self-adaptive system performance modeling approach
H(applicable)	SimuLizar's self-adaptive system performance modeling approach is applicable to model the self-adaptive system. Falsification: M(model homomorphism) show that the model does not reflect the system implementation.

Table 4.4 shows the first evaluation question *Q(appicability)*, the applied metric *M(model homomorphism)*, and our hypothesis *H(applicable)*. Our first evaluation ques-

tion is: *Is SimuLizar's modeling approach applicable to create a performance model of a self-adaptive system?* The metric to assess this question is the homomorphism of the model and the system implementation with respect to the architecture, i. e., *M(model homomorphism)*. Our hypothesis, *H(applicable)*, is that our performance modeling approach is applicable to model a self-adaptive system. The falsification of this hypothesis is possible, i. e., if the model does not reflect the system implementation, the model is not homomorph to the implementation.

Table 4.5.: Question 2: Limitations

Q(limitations)	**What are the limitations of SimuLizar's self-adaptive system performance modeling approach with respect to our modeling requirements?**
M(collection of issues)	Whenever a feature is missing to model the performance of the Znn.com system, this issue is collected. Issues are rated either *critical* or *non-critical* depending on whether the missing features is part of our modeling requirements or not.
H(only non-critical issues)	SimuLizar's performance modeling approach has only some non-critical issues. Falsification: M(collection of issues) is empty or contains critical issues.

Table 4.5 shows our second evaluation question: *What are the limitations of SimuLizar's self-adaptive system performance modeling approach?*. This question is used to check whether all of our modeling requirements are fulfilled, to detect open issues of our performance modeling approach, and to identify directions for further improvements. The metric to assess the applicability of our performance modeling approach is *M(collection of issues)*, i. e., a collection of missing modeling features, which are rated either *critical* or *non-critical*. An issue is rated critical if the missing feature is required by our modeling requirements MR1 to MR11. Otherwise, the issue is rated as non-critical. The issues are collected when applying our performance modeling approach to model the Znn.com system. Our hypothesis is, that our approach will indeed have some non-critical issues as stated in *H(only non-critical issues)*. Again, this hypothesis is falsifiable, i. e., if *M(collection of issues)* is empty or contains critical issues, the hypothesis is false.

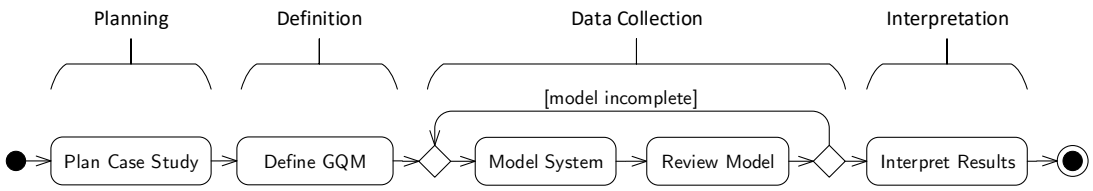


Figure 4.27.: Evaluation process.

4.8.2. Setup

In order to answer our evaluation questions, we set up a case study. In this case study, a computer science Master student at the Paderborn University got the task to apply our self-adaptive system modeling approach to model a self-adaptive system. The student had profound knowledge in software architecture modeling and some experience in software engineering. His task was to model the Znn.com system, which we introduced in Section 1.1 and used as a running example in this chapter. We chose the Znn.com system for our case study for three reasons: First, an implementation of the Znn.com system is available online [CS16]. The availability of an implementation is a prerequisite for our evaluation in order to be able to assess Metric *M(model homomorphism)*. Second, it already has been used for the evaluation of the modeling language Stitch [CG12] and the evaluation of the Rainbow framework [CGS09]. Finally, the system is well documented, as it has been used in research as an example for self-adaptive systems in various publications already [CGS09, CG12, WSG+13, SAV+16].

Figure 4.27 shows the process of our case study mapped to the four phases of the GQM approach [vBCR02]: *planning*, *definition*, *data collection*, and *interpretation*. In the initial planning phase, we established the team for the evaluation and planned the case study (Plan Case Study). The evaluation team consisted of the above mentioned computer science Master student and three performance engineering experts of our research group. In the *definition* phase, we defined the evaluation goal, questions, and metrics (Define GQM) that we presented in this section. Next, in the *data collection* phase, the student modeled the Znn.com system using our performance modeling approach (Model System). During this process of modeling the system, the model was reviewed during weekly meetings (Review Model). Finally, a complete version of the Znn.com model was reviewed by the group of performance engineering experts in order to assess

Metric M(model homomorphism), i. e., assess whether the model is homomorph to the system implementation. Additional to this review, the student collected limitations of our modeling approach, i. e., *M(collection of issues)*, whenever a feature was not available that was required to correctly model the Znn.com system. The collection of the issues was discussed and reviewed during the weekly meetings with at least one performance engineering expert. In the final *interpretation phase* the results of the case study were interpreted with respect to our evaluation goal (Interpret Results). We summarize these results in the next subsection.

4.8.3. Results

The complete Znn.com model that is the result of the data collection phase can be found in Appendix I. During the data collection phase, the performance engineering experts accepted the Znn.com model as a homomorphic representation of the implementation in terms of the architecture.

During the *data collection* phase of the case study, the participant also collected issues of our modeling approach. The participant listed one non-critical issue and no critical issues. The participant observed that the implementation of the Znn.com system drops requests in case of an overload situation when all other reconfiguration strategies fail, e. g., all available application servers are at maximum load. However, this cannot be modeled with our modeling approach. The participant rated the issue that it was not possible to model that the Znn.com system drops incoming requests as a non-critical issue since it was not part of the modeling requirements.

4.8.4. Discussion

Our case study shows that both of our hypothesis, *H(applicable)* and *H(only minor issues)*, hold. This means, that our performance modeling approach is applicable for performance modeling of self-adaptive system architectures but still has some minor limitations. Overall, we successfully validated the applicability of the modeling approach (Level II validation).

Our first evaluation question, *Q(applicability)*, was: *Is SimuLizar's modeling approach applicable to create a performance model of a self-adaptive system?* We can answer this questions with "yes". The evidence for the positive answer can be implied from the analysis of the requirement fulfillment.

The reviewers accepted the model as a homomorphic model of the real implementation. Thus, first, we can conclude that the presented view types fulfill the requirements MR5 to MR11 which describe the scope of the performance model. We can conclude that software engineers can specify a complete self-adaptive system performance model. Second, we can conclude from the case study that the self-adaptive system performance model can be created at design-time (MR1) with the help of our implementation, SimuLizar Bench (MR4), that we present in Chapter 6. Third, the presented modeling approach supports the separation of concerns (MR2) by splitting the self-adaptive system performance model into distinct viewpoints and views and distinct modeling responsibilities of the roles. Finally, the specification of resource-demanding behavior also enables an analysis of the modeled self-adaptive system architecture with respect to its scalability and elasticity (MR3). The results of the scalability and elasticity prediction within the case study are presented in discussed in Chapter 5.

Our second evaluation question, *Q(limitations)*, was: *What are the limitations of SimuLizar's self-adaptive system performance modeling approach with respect to our modeling requirements?* Our case study showed that there is only a non-critical issue concerning the presented modeling approach.

The identified non-critical issue means, that our modeling approach lacks the capability of expressing that incoming user requests are dismissed, i. e., the request queue is purged. This issue stems from the limitation of Palladio's underlying simulation engine, which is based on a queuing network simulation. In queuing network simulations, it is not intended to purge a queue. Hence, this capability is also not reflected within the Palladio Component Model as well as our modeling approach. No critical issues were found in the case study, which is further evidence that our modeling approach fulfills all of our modeling requirements.

4.8.5. Threats to Validity

In this section, we discuss the threats to the validity of the case study results and the conclusions we have drawn from these results.

We can identify two types of threats to the validity of our case study results and conclusions. First, the *dependency of the presented approach to the Palladio Component Model* poses a threat to the applicability of our modeling approach. Second, two decisions within the case study design pose a threat to the validity of the case study results: the *selection of the modeling case* and the *selection of the participant*.

Dependency on PCM

The reuse of the Palladio Component Model as a basis for our modeling approach poses a threat to the validity of our case study result. First, the PCM could contain defects that lead to incorrect conclusions from the case study results. For example, the semantics of model elements like components and interfaces could have been implemented incorrectly in PCM. However, the PCM and the Palladio approach were validated by S. Becker in [BKR09]. Second, the PCM could have been applied incorrectly in our modeling approach such that correctly implemented features in PCM are incorrectly used in our modeling approach. However, we conducted several design reviews and code reviews during the implementation of our modeling approach for various components. Hence, we can assume that the incorrect application of PCM features would have been detected in these reviews.

Selection of the Modeling Case

The second threat is about the selection of the modeling case, i. e., the Znn.com system. The Znn.com system is a synthetic system from academia and no real system from industry. Thus, it could potentially be the case that self-adaptive systems in industry are designed and implemented in an essentially different way from the design and implementation of the Znn.com system. However, we assessed the risk of this threat as low. In general, it is possible that the answers to our evaluation questions would be different

if the case study were applied to a different system. Due to the lack of well-documented and available self-adaptive systems, the evaluation was limited to the Znn.com system. Still, we see the need for more evaluations, especially in an industrial context. For this purpose, our case study design can be reused also in industrial case studies.

Selection of the Participant

The third threat is about the selection of participant within the case study. According to Prechelt computer science students in an advanced stage of the study, e. g., Master students, are suitable participant group for case studies in the context of software engineering. The skills of Master students are comparable to professional software engineers, since their software engineering training is almost finished and hence are classified as professionals soon. An advantage of students over professional software engineers is also that professional software engineers are often highly specialized and thus do not match the required profile of the case study to be performed. [Pre99]

Still, there was a risk that the student may have been biased towards a positive result of the validation due to his involvement in our research group that contributed to the modeling approach. Furthermore, the author of this thesis was one of the performance engineering experts within the case study that decided whether the model is homomorphic to the real implementation. However, the two other performance engineering experts could have outvoted the third performance engineering expert. Thus, unbiased decisions of the expert group was ensured.

In summary, we see only a low risk for the invalidation of our case study. However, since the case study was conducted in an academic context only, we recommend repeating and extending the validation in an industrial context.

4.9. Conclusion

In this chapter, we presented modeling requirements that specify the scope and necessary properties that a modeling approach to model performance properties of self-

adaptive systems must implement. We discussed existing modeling approaches according to these requirements and found that none of the discussed approaches fulfills all requirements. However, we identified the Palladio Component Model as the best basis for a new modeling approach that fulfills all of our requirements. Hence, we presented SimuLizar's self-adaptive system performance modeling approach, which is based on PCM. SimuLizar closes PCM's gaps and fulfills all of our modeling requirements. We evaluated the applicability of the presented modeling approach within a case study. The case study showed that our approach is applicable to self-adaptive systems like the Znn.com system. However, further empirical studies have to be conducted in order to show the applicability and efficiency of the approach in industrial software engineering projects.

"Prediction is very difficult, especially if it's about the future."

Niels Bohr

5

Scalability and Elasticity Prediction Methods

Contents

5.1. Scientific Contribution 112
5.2. Prediction Method Requirements 113
5.3. Related Work 115
5.4. Prediction Methods Overview 121
5.5. Self-Adaptive System Formalization 123
5.6. Service Level Objective Formalization 129
5.7. Scalability Prediction 145
5.8. Elasticity Prediction 152
5.9. Evaluation 158
5.10. Conclusion 168

A fluent performance with low response times is an important business success factor for large-scale business information systems [SW03]. Still, many software engineering projects fail since performance objectives are not achieved. The causes for these failures often are wrong decisions in the early phases of the software engineering process [SW03]. "Fixing these [performance] problems is costly and causes schedule delays, cost overruns, lost productivity, damaged customer relations, missed market windows, lost revenues, and a host of other difficulties. In extreme cases, it may not be possible to fix performance problems without extensive redesign and re-implementation. In those cases, the project becomes either an infinite sink for time and money, or it is, mercifully, canceled. Performance cannot be retrofitted; it must be designed into software from the beginning" [SW03]. Hence, early assessment of performance properties in the design phase of the software engineering process is crucial to avert project failures due to performance issues.

There have been some approaches to predict performance properties early in the software engineering process. The original approaches were based on analytical models, such as queuing networks (QN) or stochastic Petri nets [Smi90, Jai91]. These approaches required software engineers to manually derive analytical models that reflect the software architecture. Only with more recent model-driven approaches [DN02, BDIS04, Koz10], the barriers for software engineers to integrate performance predictions into the software engineering process are dismantled [BDIS04]. These model-driven approaches provide automatic transformations from software architecture models to analytical models. Thus, these model-driven approaches facilitate the assessment of performance properties early at design-time with manageable additional effort for software engineers.

However, also these model-driven approaches rely on the assumption that the system context is not uncertain but completely known and that the software architectures are non-adaptive [CdLG+09b]. Hence, for modern software system like cloud computing systems, these approaches are not feasible anymore. Modern software architectures are developed for uncertain contexts and are self-adaptive, i. e., the context of the system is subject to change and the system's architecture adapts during operation to the changing context. Consequently, software engineers need methods to assure the quality of the software architectures in the presence of uncertainty and change [CdLG+09b]. Tra-

ditional model-driven software performance approaches are not applicable to achieve this goal since these approaches assume complete knowledge of the system's context. In these approaches, a single, static analytical model is generated by a one-time model transformation of the software architecture [GMR09]. Context changes and reconfigurations of the software architecture are not reflected.

Relevant performance properties for self-adaptive systems have been partly identified, but are not well defined yet [JW00]. For example, scalability has been identified as "an increasingly important aspect of today's software systems" [SW03], but existing metrics to quantify the scalability of a software architecture still neglect the key characteristics of self-adaptive systems [JW00], i. e., the uncertainty of the context is neglected. However, according to Reussner scalability is dependent on context variability [RF08] and thus context variability and its consequent uncertainty at design-time must be explicitly taken into account in the definition of concrete scalability metrics. The situation is even worse for elasticity. An attempt for a comprehensible definition of elasticity has only recently been made by Herbst et al. [HKR13]. Metrics and prediction methods to obtain these metrics early at design-time, however, have not been defined yet.

Due to the lack of defined metrics and prediction methods for scalability and elasticity, design-time prediction of scalability and elasticity is still not possible for self-adaptive systems.

In this thesis, we provide formal metric definitions for scalability and elasticity that both characterize the quality of the software architecture's self-adaptive layer. The definitions are based on our formalization for self-adaptive systems that we introduce in this chapter as well. The self-adaptive system formalization defines the semantics of our self-adaptive system performance model that we introduced in Chapter 4. The self-adaptive system performance model serves as input for our analysis methods that allow software engineers to assess the scalability and elasticity metrics at design-time. The provided methods are part of SimuLizar and are implemented in SimuLizar Bench. Thus, the provided methods are integrated into SimuLizar's model-driven performance engineering method.

We successfully evaluated our prediction methods on the Znn.com system. The evaluation shows that our prediction methods are applicable to predict scalability and elastic-

ity properties early in the software engineering process. Thus, our prediction methods can help to avert project failure due to performance issues and to reduce development time and costs.

In this chapter, we first give an overview of our scientific contributions in Section 5.1. Second, in Section 5.2, we specify the requirements for the model-driven scalability and elasticity prediction methods. We discuss existing performance prediction methods with respect to these requirements in Section 5.3. The discussion of existing performance prediction methods helps us to identify the best candidates to base our prediction methods on and reveals which issues in related work are still open and need to be addressed in our methods. Subsequently, we give an overview over our methods for the prediction of scalability and the prediction of elasticity in Section 5.4. We provide a formalization of self-adaptive systems in Section 5.5 and a formalization of service level objectives in Section 5.6. The formalizations build the basis for our prediction methods that we present in the subsequent sections in more detail. In Section 5.7 we present our scalability prediction method and in Section 5.8 we present our elasticity prediction method. For each of the two methods, we provide a formalization, metrics, and outline their implementation as well as underlying assumptions and limitations. Additionally, we discuss both prediction methods with respect to our requirements. Subsequently, we describe the evaluation of our prediction methods in Section 5.9. Finally, in Section 5.10, we draw conclusions for the presented prediction methods.

5.1. Scientific Contribution

The scientific contributions in this chapter can be summarized as follows:

- We provide two sets of requirements for design-time scalability and elasticity prediction methods. The first set comprises general requirements to define mandatory properties for prediction methods as defined by Balsamo et al. [BDIS04, BM04]. Additionally, we define specific requirements that address the prediction of scalability and elasticity of self-adaptive systems. We base these specific requirements on the requirements for prediction methods for self-adaptive systems by Böhme [BF08], Grassi [GMR09], and Reussner [RF08].

- We systematically review existing model-driven performance prediction methods based on our requirements for design-time scalability and elasticity prediction of self-adaptive systems. An initial version of our systematic literature review has been published in [BLB12]. The initial systematic literature review was focused on complete engineering approaches. In contrast, the systematic literature review that we present in this thesis is more narrowly focused on model-driven performance engineering approaches for self-adaptive systems.

- We present a novel formalization of self-adaptive systems and the graded achievement of service level objectives in self-adaptive systems. The formalization is based on fuzzy logic [KY95] and fuzzy sets [Zad65] and serves as the framework for a sound definition of scalability and elasticity metrics and prediction methods.

- We define scalability and elasticity metrics for self-adaptive systems. The metrics are based on our formalization of self-adaptive systems and enable the assessment and comparison of self-adaptive system architectures.

- We provide methods to predict scalability and elasticity properties of self-adaptive systems at design-time. A self-adaptive system performance model, as introduced in the previous chapter, serves as input for both methods. Finally, we show that our prediction methods completely fulfill our requirements.

5.2. Prediction Method Requirements

We motivated the need for model-driven prediction methods that enable the prediction of scalability and elasticity properties of self-adaptive systems early at design-time in the introduction of this chapter. In this section, we provide a set of requirements that shall be fulfilled by these prediction methods.

Our prediction method requirements are split into two sets. The first set contains general requirements for prediction methods that address the applicability of the methods by software engineers. The second set contains requirements that are specific for scalability and elasticity prediction methods and self-adaptive systems.

5.2.1. General Requirements

In this first set of *general requirements*, we specify requirements for mandatory properties for prediction methods that shall support software engineers to predict quality properties of software systems at design-time. The requirements are based on the requirements for performance evaluation tools by Balsamo et al. [BDIS04, BM04].

PR1 Design-Time: The prediction methods shall be applicable at design-time by a software engineer, i. e., in the phase in which the software architecture is created and performance-critical design decisions are made [SW03]. Consequently, the applicability of the prediction methods at design-time is required.

PR2 Model-Driven: The prediction methods shall be model-driven, i. e., architectural models of the self-adaptive system shall be the input of the analysis approach. It shall be avoided that the software architect has to manually create a second model from scratch, e. g., an analytical model, just for the purpose of the scalability and elasticity predictions. Thus, also inconsistencies between different models due to manual translation are avoided [BDIS04].

PR3 Integrated Tool Chain: The prediction of scalability and elasticity of self-adaptive systems shall be integrated in a modeling tool chain. Required model-transformations for the prediction shall be executed with as few interaction of the software engineer as possible, or even be fully automated [BM04, GMR09].

5.2.2. Requirements for Scalability and Elasticity Prediction

In the second set of *requirements for scalability and elasticity prediction*, we specify specific requirements for the prediction of scalability and elasticity of self-adaptive systems. This second set of requirements is based on requirements by Böhme [BF08], Grassi [GMR09], and Reussner [RF08].

PR4 Formally Defined Metrics: Prediction methods for the assessment of scalability and elasticity properties shall rely on formally defined metrics. A formal

definition of the metrics allows to assess the quality of the metrics, e. g., the validity and reliability [BF08].

PR5 Prediction Method: To assess scalability and elasticity properties of self-adaptive systems, methods for each property shall be provided. The methods shall enable to obtain metrics that quantify the respective property of the system [RF08].

PR6 System Configuration: The prediction methods shall take the self-adaptation layer into account. That is, the prediction methods shall be applicable to the whole system architecture configuration space and not only to single system architecture configurations [GMR09].

PR7 System Context: In addition to the system architecture configuration space, the prediction methods shall take the system context, e. g., workload scenarios, into account [RF08]. The context change drives the self-adaptation and shall be explicitly addressed in the prediction methods.

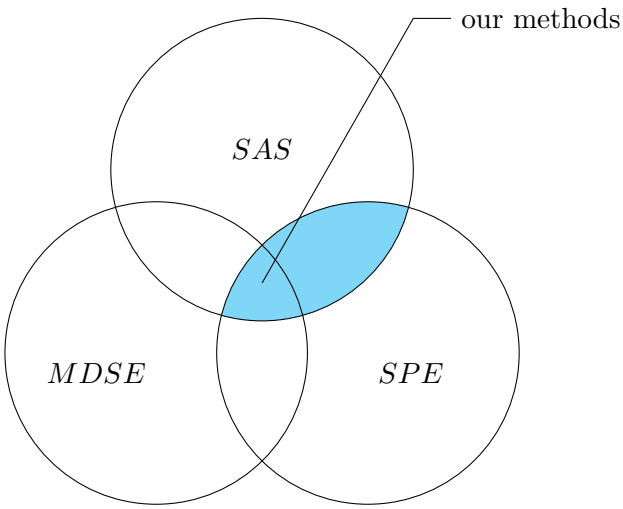
5.3. Related Work

Model-driven scalability and elasticity prediction for self-adaptive systems at design-time is a research topic that is associated to three research areas, as illustrated in Figure 5.1. First, it is associated to *model-driven software engineering (MDSE)*. In model-driven software engineering the model of the software to be built is the central artifact for the engineering process and thus is the starting point for all other artifacts and analyses [SVC06]. Second, model-driven scalability prediction and elasticity prediction is associated with *self-adaptive systems (SAS)* engineering. Self-adaptation has been identified as its own engineering concern and thus yielded its own engineering and analysis approaches [CdLG+09b]. Third, it is associated with *software performance engineering (SPE)*. Scalability and elasticity are commonly considered as performance properties of systems that rely on a variable amount of (virtual) resources, like in cloud computing [BLB15].

We applied the guideline for systematic literature reviews by Kitchenham [KDJ04, KC07] to find prediction methods in the intersection of three research areas that fulfill our requirements. However, since there is only few research that combines all three research areas, we broadened the scope of our survey and also include work in the intersections of only two research areas. Specifically, we include related work in the intersection of self-adaptive systems and software performance engineering, as illustrated in Figure 5.1. We decided to include this intersection, since we expect that we can find candidates that can be extended to fulfill all of our requirements. The intersection of self-adaptive systems and model-driven engineering and the intersection of model-driven engineering and software performance engineering were already included in the literature review of related work for our performance modeling approach in Section 4.3.

The survey question, which guided our systematic literature review was: *Which of the existing methods fulfills the most of our prediction method requirements?* We conducted the literature survey in the time from October 2011 to April 2016 with the search engine Google Scholar. We selected related work, according to the guideline in two steps. First, we included all papers that can be found on Google Scholar by searching for a combination of our predefined keywords. The keywords were combinations of the three research areas "model-driven (software) engineering", "self-adaptive systems", and "(software) performance engineering", with and without the word "software" as indicated by the brackets. Second, we scanned the abstracts of all results and filtered out all papers that did not match our acceptance criteria. We accepted all papers that indicated that the paper presents a performance prediction approach for self-adaptive systems.

Based on our prediction method requirements, presented in the previous section, and our findings in our initial survey [BLB12], we have created a feature model, see Figure 5.2. The feature model specifies required features for scalability and elasticity prediction methods for self-adaptive systems and to answer the survey questions of our literature review. Mandatory features in the feature diagram are features that we also identified in our set of requirements PR1 to PR7. All other features are optional and help to classify the surveyed approaches. Thus, any software performance engineering method that fulfills all of our prediction requirements is also a valid configuration of our feature model. Again, we highlighted the configuration of our own prediction methods that we



Legend
SAS = Self-Adaptive Systems; MDSE = Model-Driven Software Engineering;
SPE = Software Performance Engineering; ■ = Related Work

Figure 5.1.: Venn diagram of related work.

present in this thesis in the feature model by coloring the selected features in gray. We do not discuss related prediction methods that are based on the modeling approach we present in this thesis. Instead, we discuss follow-up work in Section 7.2.

Figure 5.2 shows our classification for related prediction methods. We classify related prediction methods according to the methods' modeling paradigms, their application time, and their applicability. This classification also reflects our first three general requirements PR1 to PR3 for the prediction methods. We first give a brief overview over the classification of the surveyed prediction methods. Second, we discuss for each of the remaining requirements PR4 to PR7 whether the requirement is fulfilled by the surveyed approaches. Finally, we summarize our findings and identify the prediction method that fulfills the requirements best. We use the best prediction method (with respect to our requirements) as a basis for our model-driven prediction method to enable the prediction of scalability and elasticity properties of self-adaptive systems at design-time.

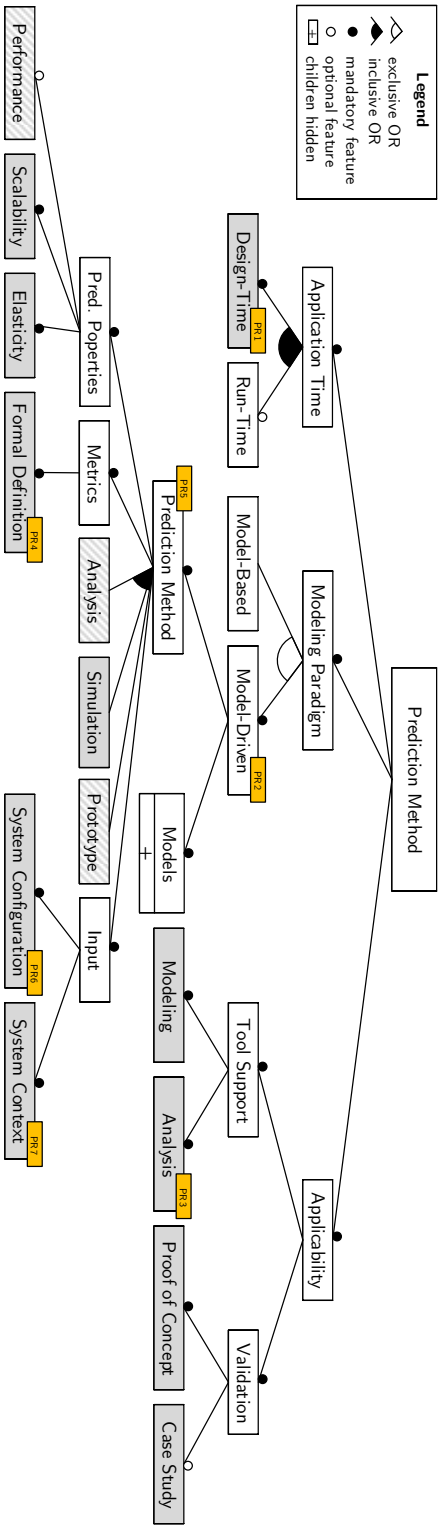


Figure 5.2.: Feature diagram for classification of related work.

Table 5.1.: Evaluation Results of Related Prediction Methods

	General			Prediction			
Approach	**PR1**	**PR2**	**PR3**	**PR4**	**PR5**	**PR6**	**PR7**
Descartes [HWBK15]	×	✓	✓	○	×	○	○
Palladio [BKR09]	✓	✓	✓	×	×	○	✓
D-KLAPER [GMR09]	✓	✓	○	×	×	✓	×
Incerto et al. [ITT15]	×	×	×	×	×	○	×
SLAstic.SIM [vMvH11]	×	✓	○	×	×	○	○
UML-Ψ [BM04]	✓	✓	○	×	×	○	×

✓= requirement fulfilled; ×= requirement not fulfilled; ○= requirement partly fulfilled;
? = unknown; - = does not apply;

Table 5.1 gives an overview of the surveyed prediction methods. From this table, it can be observed that none of the surveyed methods fulfills all requirements. Our first classification property is the application time of the prediction methods. Three of the surveyed prediction methods, Descartes [HWBK15], Incerto et al. [ITT15], and SLAstic.SIM [vMvH11], are targeted for run-time analysis., i. e., these approaches do not fulfill requirement PR1. The remaining prediction methods, Palladio [BKR09], D-KLAPER [GMR09], and UML-Ψ [BM04], are targeted for design-time predictions.

We further classify the prediction methods according to their modeling paradigm. Here we distinguish, whether the models for the prediction have to be manually created, i. e., model-based methods, or are automatically generated from higher-level architecture models, i. e., model-driven methods, as required by requirement PR2. Only the prediction method by Incerto et al. [ITT15] is model-based, as it relies on a queuing network models as input rather than on a higher-level architecture model and no automatic transformations from architectural models to these queuing network models are provided. All other prediction methods are model-driven and provide model-transformations from high-level architecture models to queuing networks, or in case of D-KLAPER, a model transformation from high-level architecture models to Petri nets.

The applicability of the prediction methods is more heterogeneous than the other classification properties. Only Descartes [HWBK15] and Palladio [BKR09] fully integrate the prediction methods into a tool chain, as required by requirement PR3.

The "Formal Defined Metrics" requirement PR4 is not completely fulfilled by any of the surveyed prediction methods, i. e., none of these methods formally define scalability and elasticity metrics with respect to our requirement. In Descartes only elasticity metrics are defined that measure speed and precision of resource leasing [HKR13] with respect to the resource demand. However, a scalability metric definition is not provided.

Requirement PR5, i. e., "Prediction Method", is neither fulfilled by any of the surveyed approaches. Rather than providing methods to predict scalability and elasticity, the surveyed approaches focus on the prediction of traditional performance metrics, like response time or utilization. All of the surveyed methods provide means to predict these traditional performance metrics. For example, the Palladio approach [BKR09] provides a model-driven method for prediction performance metrics like response time, utilization, and waiting times for non-adaptive software architectures. Even though D-KLAPER [GMR09] implements a model-transformation from architecture models to analysis models for self-adaptive systems, a method to assess scalability and elasticity metrics from these analysis models is not provided. The same is true for the approach by Incerto et al. [ITT15]. They provide an analytical prediction method for self-adaptive systems that is, however, limited to the performance metric response time, which is not applicable to measure the quality of the self-adaptation layer itself.

The "System Configuration" requirement, PR6, is at least partly fulfilled by all of the surveyed approaches. However, only D-KLAPER [GMR09] takes the whole system architecture configuration space into account. All other approaches take only a single architecture configuration into account. For the run-time targeted methods, Descartes [HWBK15], Incerto et al. [ITT15] and SLAstic.SIM [vMvH11], this architecture configuration reflects the current architecture configuration of the running system. Palladio [BKR09] and UML-Ψ [BM04] consider only a single non-adaptive system architecture configuration.

Finally, Requirement PR7, "System Context", is partly fulfilled by Descartes [HWBK15] and SLAstic.SIM [vMvH11]. The requirement is completely fulfilled by Palladio [BKR09]. Descartes [HWBK15] and SLAstic.SIM [vMvH11] take the system context for the predictions into account. In both approaches, the run-time system context is continuously monitored and serves as input for run-time predictions. In Palladio mul-

tiple system contexts can be defined in the form of usage scenarios. A prediction for a system configuration can be made for each usage scenario or also a combination of usage scenarios. All other prediction methods do not support the system context as a dedicated input for the prediction but rather mix resource consumption of the system configuration with workload of the system context.

In conclusion, we can summarize that none of the surveyed prediction methods completely fulfills all of our requirements PR1 to PR7. However, we identify Palladio as the best basis for our own model-driven scalability and elasticity prediction methods. First, Palladio completely fulfills four requirements and partly fulfills one requirement out of seven requirements in total. Second, Palladio also integrates well with our modeling approach that is based on the Palladio Component Model (PCM). Finally, just like PCM the Palladio prediction methods are implemented as open source software and are freely available, well documented, and supported by an active community [Kar16a]. The second best candidate is Descartes, which is also partly based on the Palladio implementation, but is focused on run-time performance predictions.

Consequently, we selected Palladio as the basis for our model-driven scalability and elasticity prediction methods. In order to fulfill all of our requirements, we provide extensions to Palladio, as part of SimuLizar, which focus on requirements PR4 to PR6, which are not completely fulfilled by Palladio. First, we provide a formalization and define scalability and elasticity metrics to fulfill Requirement PR4 “Formal Defined Metrics”. Second, we define and implement model-driven scalability and elasticity prediction methods to fulfill Requirement PR5. Finally, we integrate these prediction methods into our SimuLizar approach such that the complete system architecture configuration space is taken into account. Thus, also Requirement PR6 “System Configuration” is fulfilled. In the remainder of this chapter, we present SimuLizar’s formalization for self-adaptive systems and the scalability and elasticity prediction methods in more detail.

5.4. Prediction Methods Overview

As illustrated in Figure 5.3, we provide *formalizations* for a self-adaptive system and service level objectives. Based on these formalizations, we formally define the two

performance properties *scalability* and *elasticity* using the fuzzy branching temporal logic (FBTL) [MLL04]. For both properties, we define concrete *dependent metrics*, as introduced in Section 3.2, which explicitly depend on the workload of the self-adaptive system. We provide two distinct methods for the prediction of the scalability and elasticity properties and their according metrics.

Our *scalability prediction* analyzes the states of a self-adaptive system individually, i. e., it is a steady state analysis. Our *elasticity prediction* also analyzes the transitions between the states of a self-adaptive system, i. e., it is a transient analysis. The input for both of these prediction methods is a *self-adaptive system performance model* like described in Chapter 4. The output of both methods is a report about the *graded SLO achievement* that allows software engineers to assess the overall SLO achievement grade for the simulated scenario.

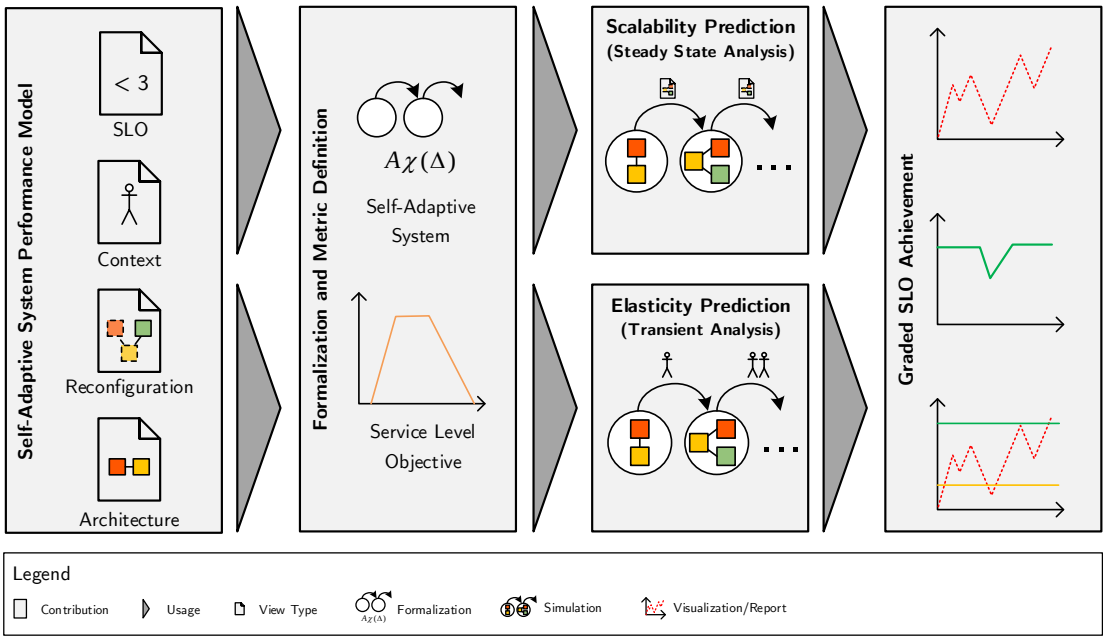


Figure 5.3.: Overview of the prediction methods

With the help of our *scalability prediction method*, a software engineer can predict if a software architecture she designed, is scalable in terms of increasing request rate in a given context scenario. For this, our prediction method will analyze each software architecture configuration that is reachable with the specified reconfigurations within

the usage context. The system is scalable if a reachable architecture configuration exists, that achieves all service level objectives.

Our *elasticity prediction method* helps software engineers to predict to which extent a software architecture is elastic, i. e., the degree the system is able to adapt to changing workloads. For this, our prediction method will analyze the performance of the modeled self-adaptive system architecture within a specified workload context. In contrast to the scalability prediction method, in this prediction method also the transitions between different architecture configurations, i. e., the reconfigurations, are analyzed. The system is elastic if the report shows that all SLOs are achieved.

We will introduce the formalization of self-adaptive systems in Section 5.5 and graded service level objective achievement in Section 5.6. Subsequently, we will describe the scalability prediction method in Section 5.7 and the elasticity prediction method in Section 5.8. For each of the two methods, we define a FBTL predicate based on our formalizations for self-adaptive systems and graded SLO achievement. Furthermore, we define for each property, i. e., scalability and elasticity, metrics and describe an implementation of how to obtain these metrics. Finally, we discuss basic assumptions and limitations of the prediction methods.

5.5. Self-Adaptive System Formalization

We described the characteristics of self-adaptive systems in Section 2.2.2 and presented a performance modeling approach for self-adaptive systems in Chapter 4. In this section, we present a formalization of a self-adaptive system that defines the semantics of our performance model. The formalization is based on our initial formalization presented in [BLB13]. Based on this formalization, we present an implementation of our scalability prediction method in Section 5.7, and an implementation of our elasticity prediction method in Section 5.8.

Our self-adaptive system formalization is split into three parts. First, we define the *structure* and *behavior* of a self-adaptive system. Second, we define the *state* of a self-adaptive system and a *state transition* function. Finally, we define the complete state

space, i. e., the *reconfiguration space*, of a self-adaptive system. For our definitions, we rely on a discrete time domain $T \subseteq \mathbb{Q}_0^+$, i. e., we interpret time as discrete points in time $t = 0 \ldots \infty$. We have choosen a discrete time domain for our formalization such that time periods we analyze are restricted, i. e., at most countably infinite.

Structure In Definition 5.1, we define a *self-adaptive system structure* $\mathcal{S}$ as a tuple that contains a set of system architecture configurations A, a set of monitored properties P, and a set of context scenarios S.

Definition 5.1 (Self-Adaptive System Structure)
A self-adaptive system structure $\mathcal{S}$ is a tuple $\mathcal{S} = (A, P, S)$, in which

- *A is the set of system architecture configurations, $A = \{a_0, a_1, a_2, \ldots\}$;*
- *P is the set of monitored properties of the system, $P = \{p_0, p_1, p_2, \ldots\}$;*
- *S is the set of context scenarios, $S = \{s_0, s_1, s_2, \ldots\}$.*

The three sets in $\mathcal{S}$ —architecture configurations A, monitored properties P, and context scenarios S— are represented by individual view types in our self-adaptive system performance model that we described in Chapter 4. The set of context scenarios S is defined via the system usage context view type and the set of monitored properties P is defined via the monitor repository view type. All architecture configurations are only implicitly defined via the initial system architecture configuration view type, the initial system deployment view type, and reconfigurations view type. The specified reconfigurations define the reconfiguration space, i. e., all reachable system architecture configurations and system deployments.

Behavior The behavior of a self-adaptive system is determined by three functions. First, the self-adaptive system monitors concrete values for the monitored properties p_i via *monitoring*, e. g., it measures the response time of requests. Second, the self-adaptive system executes *architecture reconfigurations* whenever the measurements exceed defined thresholds. Third, the self-adaptive system may face a *context scenario*

change depending on the time. These behavior functions are defined in Definition 5.2, Definition 5.3, and Definition 5.4.

Definition 5.2 (Monitoring and Run-Time Measurements)
Monitoring within a self-adaptive system is a function $\Delta(p_i, a, s, t)$ *that maps a property* p_i*, a architecture configuration* a*, a context scenario* s*, and a point in time* t *to a real number value. The real number value represents the quantification of the property* p_i *at time* t *in the given context scenario and architecture configuration, i. e., it represents a run-time measurement. When no measurement is available, e. g., in the initial state of a system, the value is undefined, i. e.,* $\Delta(p_i, a_0, s, t) = undef$.
The signature of $\Delta(p_i, a, s, t)$ *is* $P \times A \times S \times T \to \mathbb{R} \cup \{undef\}$;

- *We denote the run-time measurements of all elements* $p_i \in P$ *at time* t *with* $\Delta_P(a, s, t) = \langle \Delta(p_0, a, s, t), \Delta(p_1, a, s, t), \Delta(p_2, a, s, t), \ldots \rangle$;
- $M(a, s)$ *is the set of all run-time measurements* $\Delta_P(a, s, t)$ *for a given architecture configurations and context scenario,*

  $M(a, s) = \{\Delta_P(a, s, 0), \Delta_P(a, s, 1), \Delta_P(a, s, 2), \ldots\}$.

Definition 5.2 defines the *monitoring* behavior and *run-time measurements* of a self-adaptive system. Monitoring of each property p_i of a self-adaptive system is represented by a function $\Delta(p_i, a, s, t)$ that maps the property p_i, an architecture configuration a, and a context scenario s for a point in time t to a real number value. When no measurement is available, the value of $\Delta(p_i, a, s, t)$ is $undef$ (undefined).

In our Znn.com example, the mean response time property p_{mrt} can be monitored via a function $\Delta(p_{mrt}, a, s, t)$ that measures the mean response time (p_{mrt}) at time t.

The monitored values, i. e., run-time measurements, for all elements $p_i \in P$ for a point in time t are composed in vector $\Delta_P(a, s, t) \in M(a, s)$, where $M(a, s)$ is the set of all run-time measurements. As described in Chapter 6, the function $\Delta(p_i, a, s, t)$ is implemented as an extension to Palladio and the run-time measurement elements $\Delta_P(a, s, t)$ are implemented as a run-time measurement model in SimuLizar Bench.

Definition 5.3 (Architecture Reconfiguration) *An architecture reconfiguration of a self-adaptive system is a function $\alpha(a, \pi(m)) \in A$, where $m \in M(a, s)$ and π is a real number variable constraint vector, see Definition II.6 in Appendix II. The signature of alpha is $A \times Boolean \rightarrow A$, i. e., α maps a self-adaptive system architecture configuration and a real number variable constraint vector (over a run-time measurement) to a self-adaptive system architecture configuration.*

Architecture reconfigurations are represented by a function $\alpha(a, \pi(m))$, as defined in Definition 5.3. The function α maps the combination of an architecture configuration $a \in A$ and a real number variable constraint vector π to an architecture configuration $a \in A$. If at least one real number variable constraint in the vector evaluates to true, an architecture reconfiguration, as described by the mapping, is executed. In our Znn.com example, the function α may define a mapping from the architecture configuration with a single replica of the application tier and real number variable constraint vector $\pi = \langle \Delta(p_{mrt}, a, s, t) \geq 2.0s \rangle$ to an architecture configuration with a replicated application tier. In this example, the reconfiguration is triggered if the real number variable constraint $\Delta(p_{mrt}, a, s, t) \geq 2.0s$ evaluates to *true*. The architecture reconfigurations are implemented as model transformations in SimuLizar Bench, as described in Chapter 6.

Definition 5.4 (Context Scenario Change) *A context scenario change of a self-adaptive system is a function $\sigma(s, t) \in S$. The signature of $\sigma(t)$ is $T \rightarrow S$, i. e., σ maps a point in time to a context scenario.*

Definition 5.4 defines the *context scenario change* function that represents the change of the context scenario at run-time. The context change function maps a point in time $t \in T$ to a context scenario $s \in S$. This corresponds to a discretization of the time-dependent variations of the system usage context, as described in Section 4.6. In our Znn.com system, the context scenario may change from a low workload context scenario to a high workload context scenario at time $t = 600$, for example. Accordingly, time-dependent variations of a single usage model are mapped to discrete points in time. Context scenarios are implemented by usage models and usage evolution in SimuLizar Bench, as described in Chapter 6.

State and State Transitions Definition 5.5 defines a state $\Sigma_t \in \Sigma$ of a self-adaptive system, where Σ is the set of all states of a self-adaptive system.

Definition 5.5 (Self-Adaptive System State)
A self-adaptive system state Σ_t is a tuple $\Sigma_t = (a, s, m, t)$, in which

- *a is an element of the set of system architecture configurations, $a \in A$;*
- *s is an element of the set of context scenarios, $s \in S$.*
- *m is an element of the set of run-time measurements, $m \in M(a, s)$;*
- *t is a point in time, $t \in T$.*

A *self-adaptive system state* is, as defined in Definition 5.5, a tuple Σ_t consisting of a system architecture configuration a, a context scenario s, a run-time measurement element m, and a point in time t. The transitions between states are determined by the self-adaptive system behavior, as defined in Definition 5.6.

Definition 5.6 (Self-Adaptive System State Transition)
Self-adaptive system behavior is a labeled transition function $\rightarrow$ with labels l: $(a, s, m, t) \xrightarrow{l} (\alpha(a, \pi(m)), \Delta_P(a, s, t), \sigma(t+1), t+1)$, where $l \in \mathcal{P}(L)$ and $L = \{l_\Delta, l_\alpha, l_\sigma\}$, see Appendix II; The signature of $\rightarrow$ is $\Sigma \times \mathcal{P}(L) \rightarrow \Sigma$.

Self-adaptive system behavior is a labeled transition function $\rightarrow$ that applies the behavior functions α, Δ, and σ to the elements of a source self-adaptive system state $\Sigma_t = (a, s, m, t)$ and increases the point in time t. The resulting self-adaptive system state is the target state of the transition. The transition is labeled with the labels l_Δ, l_α, and l_σ, depending on which functions map to new elements in the state tuple, see Definitions II.1, II.2, and II.3 in Appendix II.

Reconfiguration Space With the definitions of self-adaptive system states and state transitions, we can define a self-adaptive system reconfiguration space as a labeled transition system (LTS) as in Definition 5.7.

Definition 5.7 (Self-Adaptive System Reconfiguration Space)
A self-adaptive system reconfiguration space is an LTS $\Gamma = (\Sigma, \rightarrow, \Sigma_0)$*, in which*

- Σ *is the set of self-adaptive system states;*
- $\rightarrow$ *is the labeled transition function, see Definition 5.6;*
- Σ_0 *is the initial self-adaptive system state.*

A self-adaptive system reconfiguration space is a labeled transition system in which Σ, is the set of self-adaptive system states, $\rightarrow$ is the labeled transition function, and Σ_0 is the initial self-adaptive system state. The initial state $\Sigma_0 = (a_0, m_0, s_0, 0)$ is defined via the Initial System Architecture Configuration view type, the Initial System Deployment view type, and an uninitialized run-time measurement element.

Figure 5.4 shows an example trace in a self-adaptive system reconfiguration space. The transitions between the states in the figure are labeled with the functions that trigger a state switch, see Appendix II. Initially, i. e., at time $t = 0$, the system is in state $(a_0, m_0, s_0, 0)$. The initial architecture configuration of the system is a_0 and is defined by the self-adaptive system definition. Furthermore, at time $t = 0$ the self-adaptive system's context scenario is s_0 and the measurement element m_0 is an uninitialized vector, i. e., all elements of vector m_t are undefined since no measurement has been taken at this time. Function σ may, for example, map to a context scenario s_1 at time $t_0 = 0$. Hence, the self-adaptive system will be in state $(a_0, ?, s_1, 1)$. The functions $\Delta(p_i, a, s, t)$ that return a quantification of the monitored properties p_i are updated simultaneously as well. Vector m contains a quantification for all properties p_i at time t. Thus, the self-adaptive system gets to state $(a_0, m_1, s_1, 1)$. Finally, if the real number variable constraint π of an architecture reconfiguration α evaluates to *true* in that new state at time $t_1 = 1$, e. g., $(\Delta(mrt, a, s, t) > 3s) = true$, the system architecture will be reconfigured such that the self-adaptive system proceeds to state $(a_1, m_1, s_1, 2)$.

Definition 5.7 formally defines a self-adaptive system reconfiguration space. Within this definitions, the sets A, P, and S represent the structure of the self-adaptive system. The functions α, Δ, and σ define the behavior of the self-adaptive system and the behavior of the system's context. In the next section, we provide a formalization of

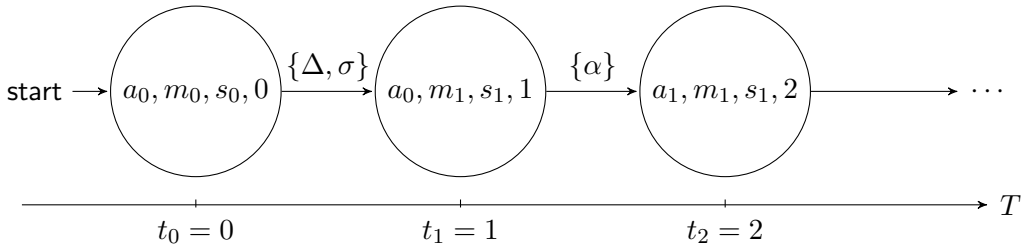


Figure 5.4.: Example trace of self-adaptive system states.

service level objectives that helps us to reason about quality properties, like scalability and elasticity, of a self-adaptive system.

5.6. Service Level Objective Formalization

Requirements for self-adaptive systems are subject to imprecision and epistemic uncertainty. The epistemic uncertainty in requirements for self-adaptive systems originates from lack of knowledge about the run-time context of the system. The imprecision originates from the inherent imprecision of (natural) language, e. g., in requirements. The uncertainty is explicitly addressed by the architecture reconfigurations in self-adaptive systems.

The architecture reconfigurations define how a self-adaptive system reconfigures its architecture to adapt at run-time in order to address its concrete contexts. Service level objectives are the run-time artifacts that steer and define the borders for this adaptation. Consequently, we provided a model for service level objectives in Chapter 4 that addresses the imprecision and allows to precisely define service level objectives with hard thresholds and with soft thresholds. Hard thresholds define borders for the self-adaptive system adaptation. Soft thresholds are used to steer the adaptation, i. e., trigger reconfigurations of the self-adaptive system.

In contrast to traditional approaches [CIL+07] that comprise a notion of binary SLO achievement, we define a notion of graded SLO achievement. That is, an SLO cannot only be fully achieved or not at all, but also to some grade. This graded SLO

achievement reflects the interval between hard thresholds and soft thresholds in which the SLOs are not fully achieved but also not fully missed.

The grade precisely reflects the solution of the trade-off between conflicting properties, e. g., cost and performance. We formally define a self-adaptive system performance model and the SLO achievement of a self-adaptive system based on fuzzy logic [KY95] and fuzzy set theory [Zad65]. We use the formal SLOs at design-time to predict SLO achievement via simulation of the modeled self-adaptive system.

Definition 5.8 (Self-Adaptive System Performance Model)
A self-adaptive system performance model is a tuple (Γ, Q)*, in which*

- Γ *is a self-adaptive system reconfiguration space;*
- Q *is the set of SLOs,* $Q = \{Q_{p_0}, Q_{p_1}, Q_{p_2}, \ldots\}$ *where each* Q_{p_i} *includes a threshold set* ϕ_{p_i} *for the quantification of property* p_i*;*
- Q_{p_i} *is an FBTL path formula over a threshold set* ϕ_{p_i}*, see [MLL04];*
- ϕ_{p_i} *is a threshold set defined as a fuzzy set with membership function* μ_{ϕ_i}*;*
- μ_{ϕ_i} *is a fuzzy set membership function* $\mu : \mathbb{R} \rightarrow [0; 1]$ *that represents the achievement of service level objective* Q_{p_i}*;*

In Definition 5.8 a self-adaptive system performance model is defined. The achievement of performance SLOs of a self-adaptive system is predicted based on this definition. We formally define a self-adaptive system performance model as a tuple (Γ, Q). In this tuple, Γ is a self-adaptive system reconfiguration space, as defined in Definition 5.7. Q is a set of service level objectives Q_{p_i}. Each SLO Q_{p_i} includes a fuzzy set ϕ_{p_i} that defines thresholds for a monitored property p_i, i. e., all acceptable values for property p_i are elements of ϕ_{p_i}.

The graded achievement of a service level objective Q_{p_i} is represented by the graded membership of a concrete value $\Delta(p_i, a, s, t)$ of a property $p_i \in P$ in the fuzzy set ϕ_{p_i}. That is, the SLO Q_{p_i} is achieved to the same grade to which the measurement $\Delta(p_i, a, s, t)$ for the property p_i is member of the SLO's fuzzy set μ_{ϕ_i}. The graded membership in a fuzzy set ϕ_{p_i} is defined by its membership function μ_{ϕ_i}.

Definition 5.9 (Graded SLO Achievement)
Let (Γ, Q) *be a self-adaptive system performance model with a set of SLOs* Q*.*

- *The SLO* $Q_{p_i} \in Q$ *is achieved in state* Σ_t *of the self-adaptive system at time* $t \in T$ *if and only if the quantification of property* p_i *is an element of the fuzzy set* ϕ_{p_i}*, i. e.,* $\Sigma_t \models Q_{p_i}$ *iff* $\Delta(p_i, a, s, t) \in \phi_{p_i}$*.*
- *The membership grade of* $\Delta(p_i, a, s, t)$ *in fuzzy set* ϕ_{p_i} *defines the graded achievement of the SLO* Q_{p_i}*.*

Definition 5.9 formally defines whether a self-adaptive system achieves its SLOs in its different states Σ_t at run-time. The achievement of an SLO Q_{p_i} by a self-adaptive system in state Σ_t is modeled by the membership grade of $\Delta(p_i, a, s, t)$ in the fuzzy set ϕ_{p_i}, i. e., $\Sigma_t \models Q_{p_i}$ iff $\Delta(p_i, a, s, t) \in \phi_{p_i}$.

In the following subsections, we describe rules how to derive the formal SLOs Q_{p_i} from requirements. The rules describe how to relax requirements and how to define thresholds ϕ_{p_i} with membership functions $\mu_{\phi_{p_i}}$. Subsequently, we will use these rules and our formal definition of a self-adaptive system to describe our scalability and elasticity prediction methods.

5.6.1. Derivation of Service Level Objectives

In SimuLizar, we refine non-functional, performance-related requirements to service level objectives. Similar as in RELAX requirements [WSB+10], we address trade-offs between requirements by relaxing single requirements. Furthermore, we introduce a graded achievement of SLOs to address the imprecision of requirements in self-adaptive systems and allow a self-adaptive system to autonomously decide trade-offs between contradicting SLOs, like costs and performance. Our graded achievement of SLOs, is based on fuzzy branching temporal logic (FBTL) [MLL04] and uses fuzzy logic [KY95] and fuzzy sets [Zad65] instead of Boolean logic.

The process to derive SLOs from requirements was briefly outlined in Section 4.4. In this section, we describe the steps in this process in more detail. Requirements are relaxed according to the two dimensions *time* and *accuracy*, as illustrated in Figure 5.5. First,

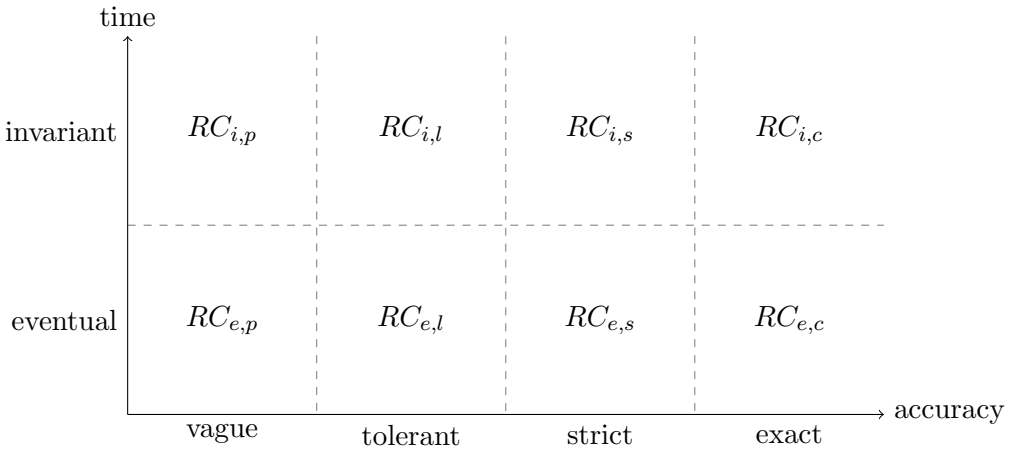
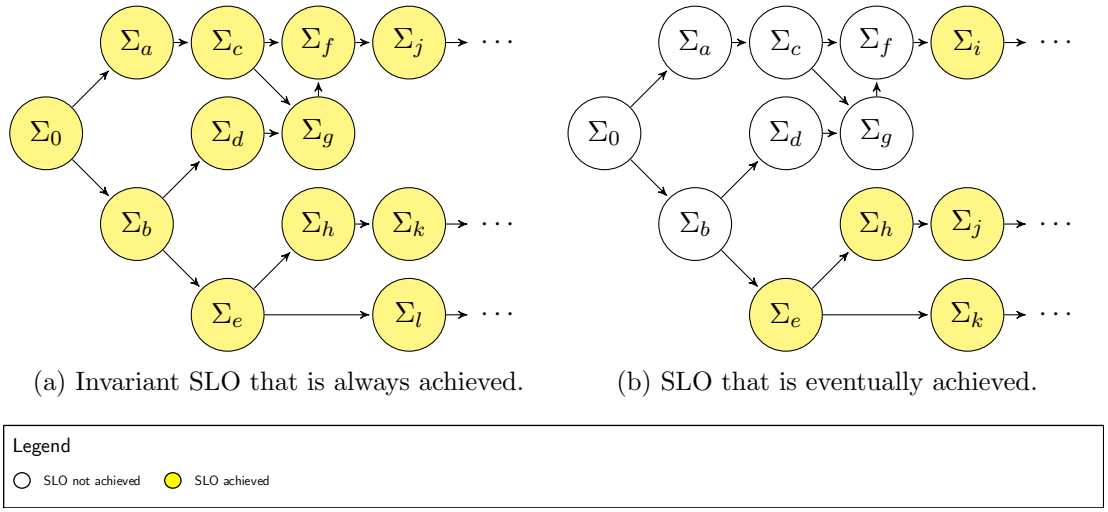


Figure 5.5.: Accuracy and time range dimensions of requirement relaxation.

to relax a certain SLO, the requirements engineer has to elicit from the stakeholders of the system in which time range a requirement has to hold. A requirement may need to hold at all times, i. e., the requirement is *invariant*, or has to hold after a certain point in time, i. e., it is an *eventual* requirement. Second, the requirements engineer has to elicit the accuracy of the concrete values of a requirement. The concrete value, like a threshold, in a requirement may only be met *vaguely*, *tolerantly*, *strictly*, or *exactly*. The relaxed requirement is then translated to a formal SLO with its according membership function.

5.6.2. Relaxation of Time

Figure 5.6 illustrates the two types in the time dimension for the relaxation of requirements. Both sides of the figure show the state space of a self-adaptive system with its initial state Σ_0 and the corresponding initial architecture configuration a_0. The nodes in the figure represent different states of the self-adaptive system with different architecture configurations a_i and the arrows in between the architecture configurations represent architecture reconfigurations α, i. e., self-adaptations. Note that only states are included in the figure with different architecture configurations, i. e., $\forall(\Sigma_x = (a_x, m_x, s_x, t_x), \Sigma_y = (a_y, m_y, s_y, t_y)) : \Sigma_x \neq \Sigma_y \Rightarrow a_x \neq a_y$, and not those that only differ in their context scenarios s_i and measurement vector m_t.



(a) Invariant SLO that is always achieved.

(b) SLO that is eventually achieved.

Figure 5.6.: Excerpts of a self-adaptive system state space.

As defined in Definition 5.10, an *invariant SLO* is achieved in all states Σ_i of a self-adaptive system Γ. In Figure 5.6, the invariant SLO is achieved in the initial state Σ_0 (with architecture configuration a_0) and all states that can be reached from Σ_0 via architecture reconfigurations α.

Definition 5.10 (Invariant SLO) *An invariant SLO Q_{p_i} shall be achieved in all states of a self-adaptive system reconfiguration space Γ. The following FBTL predicate defines an invariant SLO Q_{p_i} for the self-adaptive system reconfiguration space Γ:*
$AG(\Delta(p_i, a, s, t) \in \phi_{p_i})$

On the right side of the figure, i. e., Figure 5.6b, an SLO that is achieved eventually is illustrated. In this example, the SLO is not achieved in the initial state. Furthermore, the SLO is not achieved in all but only some of the subsequent states of Σ_0. However, the SLO is achieved in the states Σ_e and Σ_i and all of their subsequent states.

Definition 5.11 (Eventual SLO) *An eventual SLO Q_p shall be achieved in all states of a self-adaptive system reconfiguration space Γ after a defined duration d. The following FBTL predicate defines an eventual SLO Q_{p_i} for the self-adaptive system reconfiguration space Γ:*
$AX_{\geq d}(\Delta(p_i, a, s, t) \in \phi_{p_i})$, where $d \subseteq T$ is a (fuzzy) set within the time domain T.

To illustrate the relaxation of the time range in requirements, we recapitulate two requirements from our Znn.com example:

R1 The system shall serve articles requested by clients promptly. That is, the mean response time for a user request shall be less than 2.0 seconds, where the mean response time is calculated in 1 minute batches.

R2 The system's infrastructure leasing costs shall be less than USD 5.00 per hour.

Requirements R1 and R2 define invariant borders for the operation of the Znn.com system. That is, the mean response time *mrt* is required to be less than 2.0 seconds, the leasing costs *costs* less than USD 5.00 per hour. Both requirements may contradict each other, since in order to fulfill one requirement, the other requirement may not be fulfillable anymore. For example, depending on the workload scenario, a mean response time of less than 2.0 seconds may only be possible if high amounts of cloud resources are leased. Thus, the maximum leasing costs of USD 5.00 may be exceeded.

The trade-off that results from this contradiction can be resolved by relaxing at least one of the two requirements — depending on the preference of the stakeholders. For example, if the stakeholders are willing to temporarily accept higher response times in order to maintain the cost limit, Requirement R2 will not be changed and Requirement R1 can be relaxed as follows:

R1' The system shall serve articles requested by clients promptly. That is, the mean response time for a user request shall *eventually* be less than 2.0 seconds, where the mean response time is calculated in 1 minute batches.

Thus, the self-adaptive system may temporarily achieve mean response times greater than 2.0 seconds to maintain Requirement R2 without necessarily violating the (eventual) Requirement R1'.

For the operation of the system, we want to more precisely define both requirements, e. g., define an acceptable time constraint for the term *temporary*. Consequently, we derive time constraints from both requirements and define SLOs with these constraints. We use FBTL predicate, as introduced in Section 2.2.1, to define the time constraints for the service level objectives Q_{mrt} and Q_{costs}:

$$Q_{mrt} : AX_{\geq d}(\Delta(mrt, a, s, t) \in \phi_{mrt}) \tag{5.1}$$

$$Q_{costs} : AG(\Delta(costs, a, s, t) \in \phi_{costs}) \tag{5.2}$$

Formula 5.1 shows the time constraints for the formal SLO Q_{mrt} that defines thresholds for the mean response time. It uses the quantifier A to express that the SLO must be achieved in all reconfiguration paths, i. e., in the complete reconfiguration space of the system. The FBTL-specific temporal operator $X_{\geq d}$ is used to express that the SLO must hold eventually, i. e., after time d. Formula 5.2 shows the formal SLO Q_{costs} that defines a cost limit. It uses the A quantifier as well and temporal operator G to express that the SLO is invariant, i. e., must be achieved in all states of the self-adaptive system. $\Delta(mrt, a, s, t)$ is the quantification of the property mrt (mean response time), $\Delta(costs, a, s, t)$ is the quantification of the property $costs$ at time t. Functions $\mu(\phi_{mrt})$ and $\mu(\phi_{costs})$ are the membership functions of the (fuzzy) sets ϕ_{mrt} and ϕ_{costs}, that define the accepted mean response time and costs. More intuitively, the FBTL predicate 5.1 means that, eventually, the latest after time d, e. g., 10 minutes, the self-adaptive system must have reached a state (via reconfigurations) such that the leasing mean response time must be in the range defined by set ϕ_{mrt}, e. g., $\phi_{mrt} = [0; 2]$. Analogously, the set ϕ_{costs} defines the cost range, e. g., $\phi_{costs} = [0; 5]$. The concrete value for the point in time d and the sets ϕ_{mrt} and ϕ_{costs} can be either defined as crisp sets (or numbers), as in the examples above, or as fuzzy sets (or fuzzy numbers). In the latter case, the accuracy of these concrete values is relaxed and a membership function,

e. g., $\mu_{\phi_{mrt}}$, must be specified. The relaxation of accuracy is the second dimension in SimuLizar in order to relax requirements.

5.6.3. Relaxation of Accuracy

In addition to the relaxation of the time range in which a requirement must hold, in SimuLizar, the accuracy of non-functional, quantifiable requirements can be relaxed as well. We distinguish between four levels of accuracy, as illustrated in Figure 5.5: *vague*, *tolerant*, *strict*, and *exact*. Taking into account the inherent uncertainty of requirements, these different levels of accuracy reflect how accurately a value defined in a requirement shall be met.

Like presented in Chapter 4, in SimuLizar, a service level objective defines thresholds for monitored properties. More precisely, an SLO can have soft thresholds and hard thresholds. Each threshold can be either a lower threshold defining a lower boundary or an upper threshold defining an upper boundary. Soft thresholds are meant to trigger reconfigurations, i. e., if a soft threshold is exceeded at run-time, a reconfiguration of the system should be triggered. Hard thresholds should not be exceeded at all in case of invariant SLOs or only temporary for eventual SLOs.

The graded achievement of an SLO can be quantified by evaluating the according FBTL predicate of the SLO. The accuracy dimension is taken into account by different membership functions $\mu_{\phi_{p_i}}$ for the threshold set ϕ_{p_i} of an SLO for each level of accuracy. Intuitively, at the lowest level of accuracy, i. e., *vague*, an SLO Q_{p_i} is still achieved within a state Σ_t to a high grade even if the concrete measurement $\Delta(p_i, a, s, t)$ exceeds the soft thresholds extremely. At the highest level of accuracy, an SLO is not achieved at all if the concrete measurement exceed the thresholds only even a bit.

We provide template membership functions and natural language template phrases for each level of accuracy for the definition of threshold sets and their membership functions. The natural language phrases can be used to document the SLOs, e. g., as part of service level agreements. The template membership functions help domain experts to define formal SLOs that can be evaluated during the operation of a self-adaptive system. The template functions are exemplary formal definitions of the natural languages templates,

but only serve as a reference for creating membership functions. The rationale for the template functions is to capture typical natural language formulations (exact, strict, tolerant, vague) as mathematical functions. However, domain experts can define own membership functions for SLOs, if required.

For each accuracy level of an SLO, a membership function can be derived that takes into account the defined soft thresholds and hard thresholds. For the rules and illustrations, we use the following abbreviations for these thresholds:

- lt_{hard} is the hard lower threshold,
- lt_{soft} is the soft lower threshold,
- ut_{soft} is the soft upper threshold, and
- ut_{hard} is the hard upper threshold.

For all levels of accuracy, we require the hard lower threshold to be less or equal to the soft lower threshold, i. e., $lt_{hard} \leq lt_{soft}$. Analogously, we require the soft upper threshold to be less or equal to the hard upper threshold, i. e., $ut_{soft} \leq ut_{hard}$.

In the following, we define each level of accuracy with a template membership function, illustrate the membership functions as a graph, and provide template SLO phrases in natural language. The template membership function can be used either to define thresholds for the quantification of a monitored property p_i or to define the duration d after which an eventual SLO must be achieved.

Exact Accuracy

Natural Language Template 5.1 (Exact Accuracy)
`The system's` *property* `shall be exactly in the range between` lt_{hard} `and` ut_{hard}`; it shall be in no case outside of this range.`

The Natural Language Template 5.1 shows our template for exact accuracy in natural language. The template defines thresholds for a system property that shall not be exceeded at all.

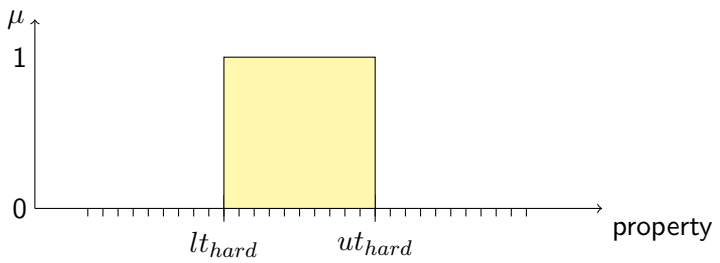


Figure 5.7.: Membership function with for SLOs with exact accuracy (non-graded achievement).

Figure 5.7 illustrates the fuzzy membership function ϕ_{p_i} of an SLO with exact accuracy. The membership function is rectangular, with value 1 for all arguments between the hard lower threshold lt_{hard} and the hard upper threshold ut_{hard}. Soft thresholds do not exist for exact accuracy or are expected to be equal to their corresponding hard thresholds. This reflects the natural language template, which states that the provided thresholds shall never be exceeded.

Function Template 5.1 (Exact Accuracy) *The membership function μ_{ϕ_p} of an SLO Q_{p_i} and threshold set ϕ_p with exact accuracy is defined as:*

$$\mu_{\phi_{p_i}}(x) = \begin{cases} 1 & \text{if } lt_{hard} \leq x \leq ut_{hard}, \text{ where } lt_{hard} = lt_{soft} \wedge ut_{hard} = ut_{soft} \\ 0 & \text{else} \end{cases}$$

The Function Template 5.1 defines the template membership function μ_{ϕ_i} of an SLO Q_{p_i} with exact accuracy. Note, that the membership function defines a crisp set, i. e., non-fuzzy set. This reflects that there is no graded achievement in terms of accuracy, i. e., if the run-time measurements $\Delta(p_i, a, s, t)$ are within the range defined by the thresholds, the SLO Q_{p_i} is achieved, otherwise the SLO Q_{p_i} is not achieved at all.

$$\mu_{\phi_{costs}}(\Delta_{costs,a,s,t}) = \begin{cases} 1 & \text{if } 0 \leq \Delta_{costs,a,s,t} \leq 5 \\ 0 & \text{else} \end{cases} \tag{5.3}$$

The membership function $\mu_{\phi_{costs}}$ that defines the thesholds for our SLO Q_{costs} in the Znn.com example is shown in Equation 5.3. If the quantification of the costs at time t,

i. e., $\Delta(costs, a, s, t)$, are within the range of 0.00 to 5.00 [USD/hour], the SLO Q_{costs} is achieved at time t. Together, with the formal definition of the time constraint for the invariant SLO, i. e., Equation 5.2, the SLO Q_{costs} must be achieved, i. e., the costs are exactly in the interval $[0; 5]$, in all states of the self-adaptive system.

Strict Accuracy

Natural Language Template 5.2 (Strict Accuracy)
`The system's` *property* `shall be strictly in the range between` lt_{soft} `and` ut_{soft}`; it may be outside of this range but never outside the range between` lt_{hard} `and` ut_{hard}`.`

The Natural Language Template 5.2 defines a natural language phrase for the strict accuracy level. In this accuracy level, the desired range for the system's property is strictly between lt_{soft} and ut_{soft}. However, the values for the property may exceed this range, but may never be lower than lt_{hard} or higher than ut_{hard}.

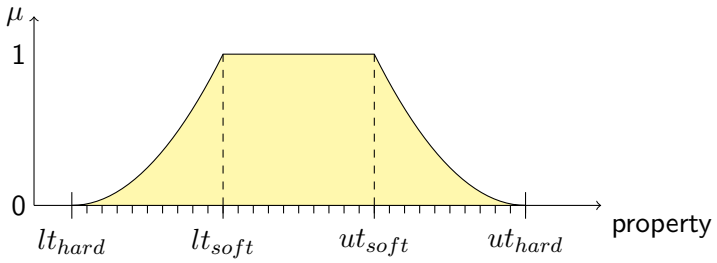


Figure 5.8.: Membership function with for SLOs with strict accuracy (quadratic achievement).

Figure 5.8 illustrates the fuzzy set membership function of an SLO with strict accuracy. In contrast to exact accuracy, we use soft thresholds and hard thresholds at this accuracy level.

In the illustrated membership function, the value is 1 for each argument in the interval between the lower soft threshold lt_{soft} and the upper soft threshold ut_{soft}. The values increase quadratically for arguments in the interval between the lower hard threshold lt_{hard} and the lower soft threshold lt_{soft}. Analogously, the values decrease quadratically

for arguments in the interval between the upper soft threshold ut_{soft} and the upper hard threshold ut_{hard}.

The shape of the curve reflects that the range between the soft thresholds is the desired range. Since this desired range shall be strictly met, the SLO achievement grade degrades quickly, i. e., quadratically, with increasing distance from the desired range.

Definition 5.12 *The membership function μ_{ϕ_p} of an SLO Q_{p_i} and threshold set ϕ_{p_i} with strict accuracy is defined as:*

$$\mu_{\phi_{p_i}}(x) = \begin{cases} \frac{1}{(lt_{soft}-lt_{hard})^2}(x - lt_{hard})^2 & \text{if } lt_{hard} \leq x < lt_{soft} \\ 1 & \text{if } lt_{soft} \leq x \leq ut_{soft} \\ \frac{1}{(ut_{hard}-ut_{soft})^2}(x - ut_{hard})^2 & \text{if } ut_{soft} < x \leq ut_{hard} \\ 0 & \text{else} \end{cases}$$

Definition 5.12 shows the template membership function $\mu_{\phi_{p_i}}$ of SLOs Q_{p_i} with strict accuracy. This membership function reflects a quadratic degradation of the SLO achievement for run-time measurement values with increasing distance from the soft thresholds. That is, the SLO achievement degrades quadratically from 1 to 0. For run-time measurements $\Delta(p_i, t)$ that are within the interval $[lt_{soft}; ut_{soft}]$, the SLO Q_{p_i} is fully achieved. For run-time measurements $\Delta(p_i, a, s, t)$ that are less than lt_{hard} or greater than ut_{hard}, the SLO Q_{p_i} is not achieved at all. The SLO achievement increases for run-time measurements in the interval $[lt_{hard}; lt_{soft}]$ quadratically and decreases for run-time measurements in the interval $[ut_{soft}; ut_{hard}]$ quadratically.

$$\mu_d(t) = \begin{cases} 1 & \text{if } 0 \leq t < 5 \\ \frac{1}{25}(t-10)^2 & \text{if } 5 \leq t \leq 10 \\ 0 & \text{else} \end{cases} \tag{5.4}$$

For our Znn.com example, we can use the template membership function for strict accuracy to define the duration d after which our SLO Q_{mrt} shall be achieved, as defined in Equation 5.1. With this definition the FBTL predicate is evaluated to 1, i. e., fully achieved, if the SLO Q_{mrt} is achieved after less than 5 minutes. The FBTL

predicate is evaluated to 0, if the SLO Q_{p_i} is achieved after more than 10 minutes. If the SLO Q_{mrt} is achieved within the range of 5 to 10 minutes, the overall achievement grade is calculated via $\frac{1}{25}(t-10)^2$ where t is the time at which the SLO is achieved.

Tolerant Accuracy

Natural Language Template 5.3 (Tolerant Accuracy)
`The system's` *property* `shall be approximately in the range between` lt_{soft} `and` ut_{soft}`; it may be outside of this range but never outside the range between` lt_{hard} `and` ut_{hard}`.`

The Natural Language Template 5.3 defines a natural language phrase for the tolerant accuracy level. In this template, the range for the system's property is approximately between lt_{soft} and ut_{soft}. Since the range is given only approximately, the values for the property may exceed this range. However, the values for the property may never be outside the range between lt_{hard} and ut_{hard}.

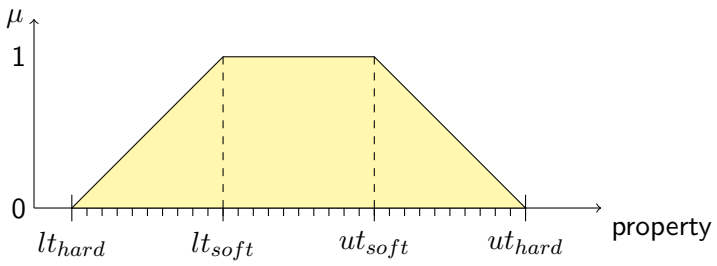


Figure 5.9.: Membership function with for SLOs with tolerant accuracy (linear achievement).

Figure 5.9 illustrates the membership function for an SLO with tolerant accuracy. Again, in this membership function we use soft thresholds and hard thresholds.

Like in the strict accuracy level, in this membership function the value is 1 for each argument in the interval between the lower soft threshold lt_{soft} and the upper soft threshold ut_{soft}. For arguments in the intervals between the hard thresholds and the soft thresholds, the values increase linearly or decrease linearly, respectively. The lin-

ear degradation of the SLO achievement grade reflects that the desired range is only approximately given, like defined in the natural language template.

Definition 5.13 *The membership function μ_{ϕ_p} of an SLO Q_{p_i} and threshold set ϕ_{p_i} with tolerant accuracy is defined as:*

$$\mu_{\phi_{p_i}}(x) = \begin{cases} \frac{1}{(lt_{soft} - lt_{hard})}(x - lt_{hard}) & \text{if } lt_{hard} \leq x < lt_{soft} \\ 1 & \text{if } lt_{soft} \leq x \leq ut_{soft} \\ -\frac{1}{(ut_{hard} - ut_{soft})}(x - ut_{hard}) & \text{if } ut_{soft} < x \leq ut_{hard} \\ 0 & \text{else} \end{cases}$$

Definition 5.13 shows the template membership function of SLOs with tolerant accuracy. The achievement of an SLO Q_{p_i} with tolerant accuracy is defined to degrade linearly from 1 to 0 with increasing distance from the soft thresholds in direction of the hard thresholds. In the interval between the lower soft threshold lt_{soft} and the upper soft threshold ut_{soft}, the SLO achievement is 1, i. e., the SLO is fully achieved. For arguments less than the lower hard threshold lt_{hard} or greater than the upper hard threshold ut_{hard} the SLO achievement is 0, i. e., the SLO is not achieved at all.

$$\mu_{\phi_{mrt}}(\Delta_{mrt,a,s,t}) = \begin{cases} \frac{1}{0.5}(\Delta_{mrt,a,s,t}) & \text{if } 0.0 \leq \Delta_{mrt,a,s,t} < 0.5 \\ 1 & \text{if } 0.5 \leq \Delta_{mrt,a,s,t} \leq 2.0 \\ -(\Delta_{mrt,a,s,t} - 3.0) & \text{if } 2.0 < \Delta_{mrt,a,s,t} \leq 3.0 \\ 0 & \text{else} \end{cases} \tag{5.5}$$

The membership function for the SLO Q_{mrt} of our Znn.com example can be defined as noted in Equation 5.5. This membership function defines a lower hard threshold lt_{hard} of 0.0 seconds and a lower soft threshold lt_{soft} of 0.5 seconds. The upper soft threshold ut_{soft} is defined as 2.0 seconds and the upper hard threshold ut_{hard} is defined as 3.0 seconds. Thus, the SLO is achieved to a certain grade for mean response times between 0.0 and 3.0 seconds. The SLO is fully achieved for mean response times between 0.5 and 2.0 seconds. The SLO achievement is linearly degraded in the intervals [0.0; 0.5] and

[2.0; 3.0] with run-time measurements $\Delta(p_i, a, s, t)$ of increasing distance from the soft thresholds in direction of the hard thresholds until it eventually becomes 0.

Vague Accuracy

Natural Language Template 5.4 (Vague Accuracy)
`The system's` *property* `shall be vaguely in the range between` lt_{soft} `and` ut_{soft}`; it may be outside of this range but never outside the range between` lt_{hard} `and` ut_{hard}`.`

The Natural Language Template 5.4 defines a natural language phrase for the vague accuracy level. The range for the system's property is vaguely defined as the range between lt_{soft} and ut_{soft}. The values for the property may exceed this range. However, the values for the property may never be outside the range between lt_{hard} and ut_{hard}.

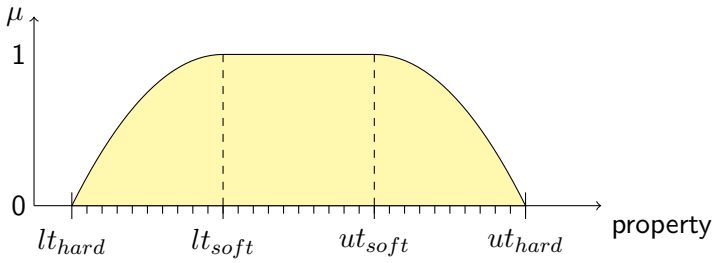


Figure 5.10.: Membership function with for SLOs with vague accuracy (negative quadratic achievement).

Finally, Figure 5.10 illustrates the membership function for an SLO with vague accuracy. Like in the previously presented membership functions, the value is 1 for arguments in the interval between the soft thresholds.

In this membership function, the value increases with a negative quadratic function for arguments in the interval between the lower hard threshold lt_{hard} and the lower soft threshold lt_{soft}. Analogously, the values decrease with a negative quadratic function for arguments in the interval between the upper soft threshold ut_{soft} and the upper hard threshold ut_{hard}. The shape of the negative quadratic function reflects the natural

language template, i. e., the SLO achievement degrades only slowly with increasing distance from the desired range.

Definition 5.14 *The membership function μ_{ϕ_p} of an SLO Q_{p_i} and threshold set ϕ_{p_i} with vague accuracy is defined as:*

$$\mu_{\phi_{p_i}}(x) = \begin{cases} 1 - \frac{1}{(lt_{soft} - lt_{hard})^2}(x - lt_{soft})^2 & \text{if } lt_{hard} \leq x < lt_{soft} \\ 1 & \text{if } lt_{soft} \leq x \leq ut_{soft} \\ 1 - \frac{1}{(ut_{hard} - ut_{soft})^2}(x - ut_{soft})^2 & \text{if } ut_{soft} < x \leq ut_{hard} \\ 0 & \text{else} \end{cases}$$

Definition 5.14 shows the template membership function for SLOs with vague accuracy. In this equation, the value is 1, i. e., the SLO is fully achieved, if the argument is in the interval between the lower soft threshold and the upper soft threshold. Like in Equation 5.12 the values in the intervals between the soft thresholds and the hard thresholds are calculated via a quadratic function, however, with inverse values, i. e., a negative quadratic function. For arguments less than the lower hard threshold lt_{hard} and arguments greater than the upper hard threshold ut_{hard} the values are 0, like in the previous membership functions.

$$\mu_{\phi_{phcf}}(\Delta(phcf, a, s, t)) = \begin{cases} 1 - \frac{1}{80^2}(\Delta(phcf, a, s, t) - 80)^2 & \text{if } 0 \leq \Delta(phcf, a, s, t) < 80 \\ 1 & \text{if } 80 \leq \Delta(phcf, a, s, t) \leq 100 \\ 0 & \text{else} \end{cases} \tag{5.6}$$

Equation 5.6 shows a final example for an SLO membership function of our Znn.com example. In this equation $\Delta(phcf, a, s, t)$ is the percentage of requests served with the highest content fidelity (phcf), i. e., multimedia, within one hour. The example membership function has no soft upper threshold. For a percentage value above 80% of the requests that are served with multimedia content fidelity, the SLO Q_{phcf} is fully achieved, as defined by this membership function $\mu_{\phi_{phcf}}$. For percentage values between

0% and 80% the achievement of the SLO increases with a negative quadratic function from 0 to 1.

The template membership functions, as presented above, can be used by domain experts to derive concrete membership functions for SLOs. In the following sections, we define the properties scalability and elasticity. Both properties characterize the quality of the self-adaptation layer. We describe methods to use a self-adaptive system model to predict scalability and elasticity of a system within a design-time simulation and discuss the assumptions and limitations of these prediction methods.

5.7. Scalability Prediction

We formally define scalability such that we can evaluate scalability at design-time, early in the software engineering process, using design-time architecture models, like our model presented in Chapter 4. For that purpose, we first recap the definition of scalability and then derive a formal definition. Subsequently, we describe a concrete metric to quantify scalability and a prediction method to obtain this metric at design-time. Finally, we discuss basic assumptions and limitations of the prediction method.

5.7.1. Formalization

Scalability was previously defined in the context of cloud computing based systems by Lehrig as "the ability of a cloud layer to increase its capacity by expanding its quantity of consumed lower-layer services." [LEB15] In the definition, the term *cloud layer* is used to refer to a software system that makes use of *lower-level services*, such as IaaS for PaaS-based software systems or PaaS for SaaS-based software systems.

Lehrig's definition states that a system is scalable if it is (potentially) able to expand its capacity, i. e., sustain increasing workload. In order to sustain increasing workload, it may make use of additional lower-level service quantities. For our purpose of formally defining scalability, we refine the definition in four aspects. First, we refine term *increase capacity* in this definition as the ability to *sustain workload variations* without missing

defined service level objectives. Second, we add a temporal dimension by adding that the system must be able to increase its capacity *eventually*. Third, we expand the term scalability in terms of the direction of scaling. We define, that a system is scalable, if it is able to increase its capacity but also to decrease its capacity. That is, a scalable system can increase or decrease its capacity without missing its service level objectives. Finally, we relax the definition by not restricting the means how to achieve the scalability. Instead of restricting this to making use of additional lower-level services, we refine the means to be any *self-adaptive* action. Thus, we can formulate our refined definition like in Definition 5.15.

Definition 5.15 (Scalability) *Scalability is the ability of a system to eventually adapt its capacity to workload variations by self-adaptation, without missing defined service level objectives.*
We formally define scalability with the following FBTL predicate in which ϕ_{p_i} is the threshold set of Q_{p_i}:

$$\forall Q_{p_i} \in Q : EF(\Delta(p_i, a, s, t) \in \phi_{p_i})$$

Intuitively, the FBTL predicate in Definition 5.15 means that there must exists at least one path (starting from the initial state) in the reconfiguration space of a self-adaptive system which finally leads to a state in which the service level objective Q_{p_i} is achieved. The path leading to the state in which the SLO is achieved is determined by the self-adaptation, i. e., architecture reconfigurations, of the self-adaptive system.

According to our formal definition of scalability, we can assess whether a self-adaptive system is scalable by examining the reconfiguration space of the self-adaptive system. If a path from the initial state to a state in which the SLO is achieved exists in the state space, the self-adaptive system is scalable. The reconfiguration space of a self-adaptive system, however, may be infinite because the context scenario s_t depends on time t, like defined in Definition 5.5 and Definition 5.6. In our formalization, time is a positive integer number, i. e., $t \in \mathbb{N}_0^+$, and the set $\mathbb{N}_0^+$ is countable, but infinite. Thus, a complete exploration of all states of a self-adaptive is not feasible. However, we can still asses the scalability of a self-adaptive system within a fixed context scenario.

Consequently, we can assess the scalability of a self-adaptive system within in a fixed context scenario s_i by exploring the reconfiguration space of the self-adaptive system. Note that, if we choose a fixed context scenario s_i, the reconfiguration space is spanned by the reconfigurations, i. e., function α, and run-time monitoring, i. e., function Δ, only. We can assume that this reconfiguration space is finite and hence the state space can be fully explored such that a path can be found if it exists.

In order to be able to compare the scalability of two self-adaptive system architectures, we need a means to put the scalability of each architecture on an ordinal (or better interval) scale, as described in Section 3.2. For this purpose, we define two metrics based on our formalization and describe methods to predict these metrics at design-time in the following sections.

5.7.2. Metric Definition

We described an initial scalability metric, *scalability range*, in [BLB15]. Based on our formalization, we derive two more precisely defined metrics from this initial scalability metric. First, we define *scalability load range (SLR)*, which is a system's ability to adapt its capacity to a certain load range while still achieving its SLOs. Second, we define *scalability work range (SWR)*, which is a system's ability to adapt its capacity to a certain work range while still achieving its SLOs.

Definition 5.16 (Scalability Load Range) *Scalability load range is the metric that reflects a system's ability adapt its capacity to a certain load range, e. g., a range of request rates for a fixed type of work, and still achieve its SLOs. That is, for each request rate within this range, the system achieves its SLOs. The scalability load range is defined as the maximum request rate within the load range. The base unit of scalability load range is defined as the base unit of the request rate, i. e., requests per second ($\frac{req.}{s}$).*

Consider a defined open workload WL for our Znn.com system that comprises a single SystemLevelEntryCall, like illustrated in Figure 4.15. For this workload WL, we can specify the scalability load range SLR_{WL} for our Znn.com system architecture, for

example $SLR_{WL} = 50\ req/min$. That is, the Znn.com architecture is able to adapt its capacity to load variations from 0 $\frac{req.}{min}$ to 50 $\frac{req.}{min}$ while still achieving its SLOs.

Work is the second parameter besides load that defines the workload of a system. While load can be specified in terms of a rate, the quantification of work is more complex. The work of a system is determined by its interaction with the user or other external systems, i. e., the methods that are called, the method parameters, and the input data. Still, work is often quantified by the amount of data to be processed for a given system usage context [BSL16]. For example, the amount of data can be quantified per request, i. e., in Bytes/request. Thus, we can also define another scalability metric, *scalability work range*, which can be used to assess the scalability of a system in dependency to a work range.

Definition 5.17 (Scalability Work Range) *Scalability work range is the metric that reflects a system's ability adapt its capacity to a certain work range, e. g., a range of work quantification for a fixed load, and still achieve its SLOs. That is, for each work quantification within this range, the system achieves its SLOs. The scalability work range is defined as the maximum work quantification within the work range. The base unit of scalability work range depends on the base unit of the work quantification, e. g., Bytes/request.*

For the same open workload WL of our Znn.com system that comprises a single SystemLevelEntryCall, we can also specify the scalability work range SWR_{WL}, e. g., $SWR_{WL} = 100\ Byte/req.$. That is, the Znn.com architecture is able to adapt its capacity to work variation from 0 $Byte/req.$ to 100 $Byte/req.$ while still achieving its SLOs.

With the metrics SLR and SWR, a software engineer is able to compare different self-adaptive system architectures and decide which architecture fits her needs best. We describe how to implement a method to predict the scalability of a self-adaptive system and obtain the metrics SLR and SWR in the next subsection.

Listing 5.1: Scalability Prediction Method

```
1  function determine_scalability_load_range(Γ, Q, s, low, high, step):
2    // Set request rate to mean value of lower and upper request rates
3    s_range = (low + high) / 2
4    Γ.s.request_rate = s_range
5    // Check whether SLOs are achieved for mean request rate
6    scales := scalability_analysis(Γ, Q, Γ.Σ_0)
7    // End recursion if distance between lower and upper
8    if (high - low) < step
9      return s_range
10   // Recursively search for max. request rate for which SLOs achieved
11   if scales = true
12     determine_scalability_load_range(Γ, Q, s, s_range + step, high)
13   else
14     determine_scalability_load_range(Γ, Q, s, low, s_range - step)
15
16 function scalability_analysis(Γ, Q, Σ_curr):
17   if ( ∀Q_i ∈ Q : Σ_curr ⊨ Q_i ) then
18     // if all SLOs are achieved, the system scales up to this workload
19     return true
20   else
21     label state Σ_curr as explored
22     // explore all states that are reachable via reconfigurations
23     for each architecture reconfiguration from Σ_curr to Σ_next in S do
24       if ( state Σ_next is not labeled as explored ) then
25         return scalability_analysis(Γ, Q, Σ_next)
26     return false
```

5.7.3. Metric Implementation

Listing 5.1 shows a pseudo code implementation of our scalability prediction method that determines the scalability load range metric of a self-adaptive system performance model (Γ, Q). The implementation of the scalability work range metric is analogue to this implementation and is hence not listed here.

The implementation consists of the two functions `determine_scalability_load_range` and `scalability_analysis`. The function `determine_scalability_load_range` has six parameters and returns the scalability range (in $\frac{req.}{s}$). The first three parameters are a self-adaptive system reconfiguration space Γ, a set of service level objectives Q, and a single context scenario s. The last three parameters are the lowest request rate in the given context scenario, the highest request rate in the context scenario, and a step width for the search algorithm.

The function `determine_scalability_load_range` implements a binary search for the highest request rate (at which the SLOs are still achieved) within the given load range, i. e., between `low` and `high`. To check whether the SLOs are achieved, the scalability of the self-adaptive system in a reconfiguration space Γ is analyzed, i. e., function `scalability_analysis` is called (line 6).

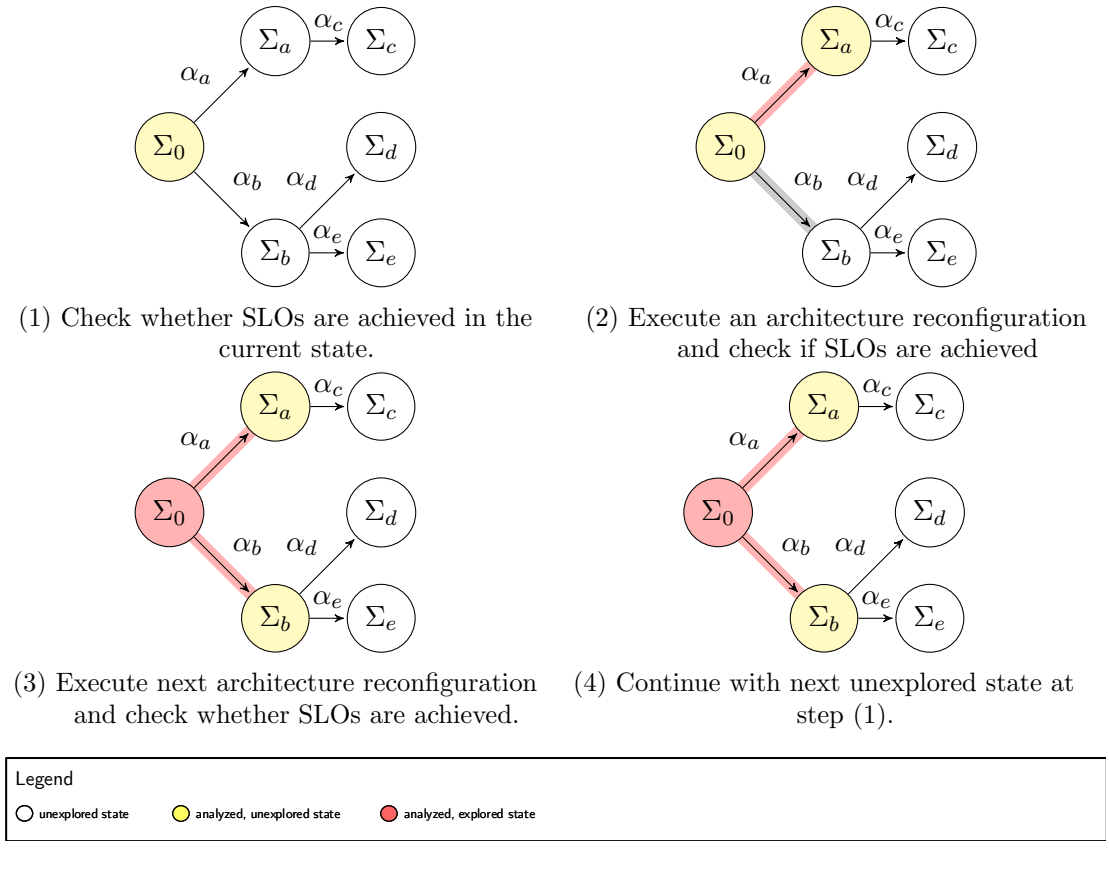


Figure 5.11.: Exploration of scalability

As outlined in Section 5.7.1, we can explore the reconfiguration space of a self-adaptive system to assess whether the system is scalable within one fixed context scenario s. This is implemented in function `scalability_analysis`. As illustrated in Figure 5.11, the function implements a breadth-first search within the reconfiguration space Γ of the self-adaptive system that is spanned by the architecture reconfigurations, i. e., function α. The breadth-first algorithm (lines 10 to 20) has three input parameters and returns a Boolean value. The input parameters are a self-adaptive system reconfiguration space Γ,

a set of service level objectives Q, and a self-adaptive system state Σ_{curr}. First, it is checked for the actual parameter system state Σ_{curr} whether the SLOs are achieved in this state (line 11). The analysis whether SLOs are achieved in one single state can, for example, be realized with a traditional model-driven software performance method like Palladio. If the SLOs are achieved, the function returns true. Otherwise, the state is labeled as explored (line 15) and the function is recursively called for all states Σ_{next} that are reachable from the current state Σ_{curr} via architecture reconfigurations, i. e., function α.

Our prediction method, implemented as described in this subsection, only terminates under some specific assumptions. We discuss these underlying assumptions and the limitations in the next subsection.

5.7.4. Assumptions and Limitations

For the scalability prediction method we presented in this section, we have two basic assumptions. First, we assume that architecture reconfigurations are cycle-free within one context scenario. However, cycles may occur due to the context scenario variations, e. g., sequences like in usage evolutions. Second, we assume that monitoring values converge for a given context scenario and architecture configuration. With these two assumptions, we can conclude that there is only a limited number of architecture reconfigurations that can be executed for a given context scenario. This is the case, since an architecture reconfiguration is according to our definition, a function $A \times \mathit{Boolean} \rightarrow A$, i. e., architecture reconfigurations are triggered if the Boolean truth value of a real number variable constraint vector ($\pi(m)$) is `true` for the current architecture configuration a. The truth value of $\pi(m)$ depends on the monitoring vector $m = \Delta_P(a, s, t)$, where this vector contains values for all monitored properties, i. e., $\Delta_P(a, s, t) = \langle \Delta(p_0, a, s, t), \Delta(p_1, a, s, t), \Delta(p_2, a, s, t), \ldots \rangle$. A measurement $\Delta(p_i, a, s, t)$ depends, for a fixed property p_i and context scenario s, on time t and the architecture configuration a, since $\Delta(p_i, a, s, t) : P \times A \times S \times T \rightarrow \mathbb{R}$. If we assume, that the monitoring values converge over time, there can only be a limited number of architecture reconfigurations as well. If, furthermore, the reconfigurations are cycle-free, only a limited number of architecture reconfigurations can be executed.

The analysis whether SLOs are achieved in a single state of the self-adaptive system, i. e., line 11 in Listing 5.1, can be realized with traditional model-driven software performance analysis methods like Palladio. In order to get reliable results for this analysis, the analysis has to run until a steady state is reached, i. e., the run-time measurements $\Delta(p_i, a, s, t)$ converge.

5.8. Elasticity Prediction

Before we formally define the second quality property of self-adaptive systems, *elasticity*, we recapitulate the definition of elasticity. We then derive a FBTL predicate and define concrete elasticity metrics that can be obtained at design-time in order to compare alternative self-adaptive system designs. Again, like for scalability, we present an implementation of our elasticity prediction method. We conclude this section with the discussion of assumptions and limitations for the presented prediction method.

5.8.1. Formalization

Elasticity is defined as "the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomous manner, such that at each point in time the available resources match the current demand as closely as possible." [HKR13] That is, a self-adaptive system has a degree, or grade, of elasticity. The system's grade of elasticity depends on its ability to adapt to changing workloads, i. e., the system's scalability. We refine the definition of elasticity in two aspects in order to derive an FBTL predicate. First, instead of limiting the objective of elasticity to matching the resource demand as closely as possible, we refine that the system must achieve all defined service level objectives. Second, we refine the term "degree" in the definition. We define that the elasticity degree depends on (1) the time it takes to achieve all defined service level objectives and (2) the overall grade of achievement of the service level objectives. Consequently, we formulate a refined definition of elasticity, as in Definition 5.18.

Definition 5.18 (Elasticity) *Elasticity is the degree to which a system is able to self-adapt to workload changes, such that it achieves all of its service level objectives to a certain grade. The degree depends on the time it takes to achieve the SLOs and on the overall grade of SLO achievement.*
We formally define elasticity with the following FBTL predicate, in which ϕ_{p_i} is the threshold set of Q_{p_i}:

$$\forall Q_{p_i} \in Q : AX_{\leq d}(\Delta(p_i, a, s, t) \in \phi_{p_i})$$

Intuitively, the FBTL predicate in Definition 5.18 means that for all path (starting from the initial state) at latest after duration d, a state must be reached in which the service level objective Q_{p_i} is achieved. The path leading to the state in which the SLO is achieved is determined by the self-adaptation, i. e., architecture reconfigurations of the self-adaptive system.

Our formal elasticity definition differs from our formal scalability definition in two major aspects. The first aspect is the scope, the second aspect is the time. While we defined for scalability that there must exist at least one path that leads to a state in which all SLOs are achieved, for elasticity the SLOs must be achieved in *all* states in all path after duration d. That is, the scope of elasticity comprises all path from the initial state of the self-adaptive system, i. e., the complete state space. Furthermore, in the elasticity definition, we specify a time constraint, i. e., duration d, after which the SLOs must hold. In contrast to this, in the scalability definition, we only specified that the SLOs must be achieved in some state of the self-adaptive system at some unspecified point in time.

We can assess the elasticity of a self-adaptive system by examining its reconfiguration space and checking whether the SLO Q_{p_i} is achieved before duration d expired. However, just like with scalability, it is not feasible to explore the complete state space due to the potential infinite number of states of the self-adaptive system, which results from the dependence on time $t \in \mathbb{N}_0^+$. Similar to the scalability prediction, however, the elasticity of a self-adaptive system can be predicted for a fixed context scenario as well.

In our scalability prediction method, all states in the reconfiguration space are explored individually. Consequently, the time it takes until monitoring values $\Delta(p_i, a, s, t)$ converge and reconfigurations, i. e., function α, are triggered, is neglected.

In the elasticity prediction, however, we are interested in exactly this time. According to our elasticity definition, the time until SLOs must be achieved in all states is limited by the duration d. Thus, the time until monitoring values $\delta(p_i, t)$ converge and reconfigurations are triggered are constrained by the duration d.

To quantify elasticity, e. g., in order to compare alternative self-adaptive system architectures, we need metrics that reflect the elasticity degree. Hence, in the following, we define two elasticity metrics based on our formalization and describe a method how to predict these elasticity metrics at design-time.

5.8.2. Metric Definitions

We described initial elasticity metrics, mean time to quality repair (MTTQR) and number of SLO violations (NSLOV), in [BLB15]. Based on the formal definition of elasticity in the previous subsection, we refine these metrics. We define *time to SLO achievment (TTSA)* as a refinement of MTTQR and *accumulated SLO achievement grade (ASAG)* as a refinement of NSLOV.

Definition 5.19 (Time To SLO Achievement) *Time to SLO achievement is the metric that reflects the duration a system requires to achieve its SLOs in a certain workload scenario. The duration is calculated as the difference from the point in time the system is in specified state, e. g., its initial state, until the point in time the system is in a state in which its SLOs are achieved. The base unit of the time to SLO achievement is defined as the base unit of time, i. e., seconds (s).*

Definition 5.20 (Accumulated SLO Achievement Grade) *Accumulated SLO achievement grade is the metric that reflects the normalized, accumulated SLO achievement grade of a system in a certain workload scenario. The ASAG value is calculated as the (normalized) integral of the SLO achievement of a system over time from the point in time the system is in a specified state, e. g., its initial state, until the point in time the system is in a state in which its SLOs are achieved. The metric has no unit, but the values are normalized and are in the interval between 0 and 1, i. e., interval [0; 1].*

The metrics defined above enable software engineers to compare alternative architectures of self-adaptive systems at design-time and select an architecture to implement based on the requirements and priorities of the stakeholders. Consider a defined usage context scenario s, like the one defined for our Znn.com example in Figure 4.15. For this usage context, we can obtain the metrics TTSA and ASAG for two alternative self-adaptive system reconfiguration spaces Γ_A and Γ_B, defined by two alternative architectures.

Let us assume, the metric values for the architecture of Γ_A are a mean TTSA of $TTSA_{\Gamma_A,s} = 60s$ and a mean ASAG of $ASAG_{\Gamma_A,s} = 0.8$. Furthermore, we assume that the concrete metric values for the architecture of Γ_B are a mean TTSA of $TTSA_{\Gamma_B,s} = 120s$ and a mean ASAG of $ASAG_{\Gamma_B,s} = 0.9$. With these metrics, we can compare both architectures of Γ_A and Γ_B.

On one hand, the architecture of Γ_A is able to achieve the SLOs in half of the time of the architecture of Γ_B for the given workload scenario s, i. e., $TTSA_{\Gamma_A,s} = 60s$ versus $TTSA_{\Gamma_B,s} = 120s$. On the other hand, the mean accumulated SLO achievement grade of architecture of Γ_B is higher than for architecture of Γ_A, i. e., $ASAG_{\Gamma_B,s} = 0.9$ versus $ASAG_{\Gamma_A,s} = 0.8$. Thus, depending on which metric we prioritize, we can identify a superior architecture. However, there might not be a single optimal architecture, but only Pareto-optimal architectures regarding the concrete metrics.

Listing 5.2: Elasticity Prediction Method

```
1  function determine_time_to_SLO_achievement(Γ,Q,s,a_start,duration):
2    for(ttsa := 0; ttsa < duration; ttsa++)
3      // Increment time in Γ and check if SLOs are achieved
4      time := ttsa
5      if (calculate_total_slo_achievment(Γ,Q,Σ_curr) > 0)
6        return ttsa
7    // SLOs have not been achieved within required duration, return ∞
8    return -1
9
10 function determine_accumulated_SLO_achievment(Γ,Q,s,a_start,duration):
11   Γ := (A,{s},P,M,Δ,α,null,a_start)
12   time := 0
13   for(ttsa := 0; ttsa < duration; ttsa++)
14     // Increment time in Γ and check if SLOs are achieved
15     time := ttsa
16     asag := asag + calculate_total_slo_achievment(Γ,Q,Σ_curr)
17   return asag / time
18
19 function calculate_total_slo_achievment(Γ,Q,Σ_curr):
20   total_slo_achievement := 0
21   total_number_of_slos := Q.size
22   // Calculate the normalized total SLO achievement for state Σ_curr
23   for (n := 0; n < total_number_of_slos; n++)
24     if (Q_n.μ_φn(Δ(p_n,a,s,curr)) = 0)
25       return 0
26     else
27       total_slo_achievement := total_slo_achievement + Q_n.μ_φn /
             total_number_of_slos
28   return total_slo_achievement
```

5.8.3. Metric Implementation

Listing 5.2 shows a pseudo code implementation of our elasticity prediction method that can be used to determine our elasticity metrics TTSA and ASAG for a self-adaptive system performance model (Γ, Q). The implementation consists of the three functions `determine_time_to_SLO_achievement`, `determine_accumulated_SLO_achievement`, and `calculate_total_SLO_achievement`.

The first two functions determine the metrics TTSA and ASAG for a given self-adaptive system performance model (Γ, Q) and in a given context scenario s. Both of these, the self-adaptive performance model and the context scenario are input parameters of both functions. Additionally, both functions require a start state a_{start} and a duration `dura-`

`tion` as input parameters. As defined in our elasticity FBTL predicate in Definition 5.18, `duration` specifies the duration after which the SLOs must be achieved.

The first function, `determine_time_to_SLO_achievement`, checks for each point in time (from 0 until *duration*) whether all SLOs Q are achieved by calling function `calculate_total_SLO_achievement` (line 6). It returns the time after which all SLOs have been achieved (line 7), i. e., the TTSA metric. If the SLOs have not been achieved within the duration `duration`, the function returns `-1` (line 9).

The second function, `determine_accumulated_SLO_achievement`, checks for each point in time the total SLO achievement at that time by calling the function `calculate_total_SLO_achievement` (line 17). Finally, the function `determine_accumulated_SLO_achievement` returns the sum of the total SLO achievement within the given duration divided by the duration (line 18), i. e., the ASAG metric. If the SLOs have not been achieved within the duration `duration`, the function returns `0` (zero), since the return value of function `determine_accumulated_SLO_achievement` will always be `0` as well.

The third function, `calculate_total_SLO_achievement`, is called by both of the two prior methods. It calculates the total SLO achievement for a self-adaptive system performance model (Γ, Q) in state Σ_{curr}. The function iterates over all SLOs Q_n with membership function μ_{ϕ_n} for the threshold set ϕ_n (lines 24 to 28). It sums up the SLO achievement grade for each SLO (line 28). The achievement grade for a single SLO is calculated via the membership function μ_{ϕ_n} of the SLO Q_n with the current monitored value of that SLO as actual parameter, i. e., $\Delta(p_n, a, s, curr)$. The function returns the sum of all SLO achievement grades divided by the number of SLOs (line 29).

The elasticity prediction method, implemented like described above, terminates only under some assumptions. We will discuss these assumptions and the limitations of our elasticity prediction method in the next subsection.

5.8.4. Assumptions and Limitations

The elasticity prediction method, which we presented in this section, will terminate under the assumption that the duration d, provided as an input, is finite. The methods to determine the elasticity metrics TTSA and ASAG terminate after the duration d has exceeded. However, to get statistically significant values, both methods should be executed k times and the mean value of the returned predeicted metric values should be calculated. The number of executions k of the methods depends on how much repetitions are required such that the run-time measurements $\Delta(p_i, a, s, t)$ — and consequently also the metrics TTSA and ASAG — converge. Hence, a general k cannot be determined, but is individual for all self-adaptive system performance models (Γ, G).

In contrast to the scalability prediction method, the analysis to which grade SLOs are achieved at a certain point in time of a self-adaptive system in state Σ_{curr}, i. e., line 25 in Listing 5.2, cannot be realized with traditional model-driven software performance analysis methods like Palladio. For an implementation of the elasticity prediction method, the self-adaptive system must be analyzed not only in a steady state but also in the transitions between its states, i. e., the underlying analysis model must reflect the self-adaptions. Hence, we implemented an interpreter for Palladio analysis models that also supports architecture reconfigurations, like presented in Section 4.7.3, and thus allows analyzing a self-adaptive system in all of its states and the transition between the states. This interpreter is part of the SimuLizar tool and is described in more detail in Chapter 6.

5.9. Evaluation

In the previous sections, we introduced scalability and elasticity prediction methods for self-adaptive systems. The goal of the presented prediction methods is to enable the assessment of self-adaptive system architectures at design-time. We formulated requirements PR1 to PR7 that need to be met in order to achieve this goal. In this section, we validate our prediction methods regarding whether the requirements are met and the goal is achieved.

As defined by Böhme and Reussner, there are three levels for the validation of a prediction method. First, a Level I validation is concerned with the validation of the predicted metrics. For this kind of validation, predictions are compared with measurements to validate the homomorphism of the predictions provided by the prediction method with respect to the reality. Second, a Level II validation is concerned with the applicability of the prediction method. That is, in this kind of validation it is evaluated whether the target group, i. e., software engineers, can reliably produce the input for the prediction method and meaningfully interpret the output of the prediction method [BF08]. Finally, a Level III validation is concerned with the benefits of the prediction method compared to other prediction methods [BF08]. This is kind of validation requires a controlled experiment in which, for example, two (or more) prediction methods are applied to the same software system in order to evaluate the benefits of one method over the other method.

Our self-adaptive system performance model that we presented in Chapter 4 is the input for our scalability and elasticity prediction that we present in this chapter. A Level II validation for our self-adaptive system performance modeling approach was presented in Chapter 4. Consequently, we present a Level I validation for our prediction methods in this chapter. Due to the limited time and resources, a Level III validation could not be conducted in the scope of this thesis.

We applied the goal question metric approach [vBCR02] to conduct a case study for the Level I validation of our prediction methods. A computer science Master student conducted the case study using the Znn.com system as the case for our validation. The student applied our prediction methods to predict the scalability and elasticity of a Znn.com system implementation. For this purpose the model that was created during the case study to validate the modeling approach (Level II validation) was reused for the Level I validation.

Table 5.2 formulates our evaluation goal using the GQM template. Our goal is to *analyze* the scalability and elasticity prediction methods, we presented in this chapter, *for the purpose of* evaluating the applicability of the prediction methods *with respect to* our prediction method requirements PR1 to PR7. The evaluation is conducted *from the viewpoint of* software engineers. The *context* of the evaluation is academic, since we use

the same model and implementation we have used for our evaluation of the modeling approach in Chapter 4.

Table 5.2.: Evaluation Goal

Analyze	SimuLizar's scalability and elasticity prediction methods
for the purpose of	evaluating the applicability of the prediction methods
with respect to	our prediction method requirements PR1 to PR7
from the viewpoint of	software engineers
in the following context:	The scalability and elasticity predictions methods are applied to a Znn.com system model.

Our evaluation showed that the presented scalability and elasticity prediction methods are applicable for the Znn.com system. In the remainder of this section, we describe our evaluation and results in more detail. First, we present our evaluation question in Section 5.9.1. In Section 5.9.2, we explain our evaluation setup. We present the evaluation results in Section 5.9.3. In Section 5.9.4, we discuss the evaluation results with respect to our prediction method requirements PR1 to PR7. Finally, we discuss the threats to validity in Section 5.9.5.

5.9.1. Question

We evaluated the applicability of our scalability and elasticity prediction methods with respect to our prediction method requirements PR1 to PR7 within a case study. For the case study, we formulated an evaluation question, a metric that helps to answer the question, and a hypothesis that we wanted to check. We then applied our prediction methods to the Znn.com system and compared our predictions to measurements from a *performance prototype*, as introduced in Section 3.3, of the Znn.com system. For this purpose, we implemented the prediction methods within an integrated modeling and analysis tool chain that we present in Chapter 6.

Table 5.3 shows our evaluation question *Q(applicability)*, the applied metric *M(prediction error)*, and our hypothesis *H(applicable)*. Our evaluation question is: *Are SimuLizar's prediction methods applicable to predict the scalability and elasticity of self-adaptive systems?* The metric to assess the applicability of our scalability

Table 5.3.: Question 1: Applicability

Q(applicability)	**Are SimuLizar's prediction methods applicable to predict the scalability and elasticity of self-adaptive systems?**
M(prediction error)	SimuLizar's prediction error compared to the measured scalability and elasticity metrics
H(applicable)	SimuLizar's prediction methods are applicable to predict the scalability and elasticity of the Znn.com system. Falsification: M(Prediction Accuracy) shows that the prediction error is over 30%.

and elasticity prediction methods is the prediction error with respect to measurements from a Znn.com performance prototype, i. e., *M(prediction error)*. Our hypothesis, *H(applicable)*, is that SimuLizar's prediction methods are applicable. According to Menasce, a prediction error of 30% concerning response time is sufficiently accurate for capacity planning [MV00]. We apply the same prediction error threshold for the acceptance of our hypothesis as well. Hence, we accept our hypothesis if the prediction error for our scalability and elasticity metrics is less than 30%.

5.9.2. Setup

In order to evaluate the applicability of our scalability and elasticity prediction methods that are part of SimuLizar, we applied the methods to the Znn.com system and compared the prediction results to measurements taken from a performance prototype of the Znn.com system.

Our evaluation is twofold. First, we applied our prediction methods manually, i. e., we built a labeled transition system based on the measurements of the performance prototype. Second, we used our implementation of the prediction methods, which we describe in Chapter 6, and compared the prediction results of our implementation with the performance prototype as well.

We generated the performance prototype from the Znn.com model that was created for the evaluation of our modeling approach. The performance prototype was generated via a model transformation with the ProtoCom add-on in the Palladio Bench, as introduced

in Section 3.3. The performance prototype consumes as many real resources, e. g., CPU time, as specified in the model.

The advantage of a performance prototype is that it can calibrated to a specific hardware and thus provides reproducible measurements according to the specification of the performance model. However, since ProtoCom cannot generate performance prototypes for self-adaptive systems, we manually implemented the self-adaptation as specified in the model.

For our evaluation, we specified a high workload context scenario, in the Znn.com system this is a request rate of approximately 4 req./s (Poisson distributed). The workload was generated for the prototype with the same parameters as for the predictions. We calibrated the performance prototype resource demands according to the Znn.com model such that a user request had a mean CPU demand of 0.3 seconds. Subsequently, we recorded 1.000 measurements and simulated 1.000 requests within SimuLizar Bench for the Znn.com system.

We calculated the scalability and elasticity metrics manually according to our definitions for both, the performance prototype as well as the prediction. For the calculation of the TTSA metric, we assumed that the SLO for the mean response time has only a hard upper threshold $ut_{hard} = ut_{soft} = 1.0s$, where the mean response time is calculated in 20s batches.

5.9.3. Results

The complete Znn.com model that was used for the case study can be found in Appendix I. During the data collection phase of the case study, scalability and elasticity metrics were predicted with our prediction methods and calculated with measurements from the performance prototype.

Table 5.4 shows the scalability and elasticity metrics for the prediction and the performance prototype. The metric values are similar for the prediction and for the measurement. However, we could only calculate the prediction error for the ASAG metric, since the precision for the SLR and TTSA metrics was not high enough to calculate

Table 5.4.: Comparison of the Scalability and Elasticity Metrics

Metric	SimuLizar Prediction	Prototype Measurement	Prediction Error
SLR	approx. 6 req./s	approx. 6 req./s	-
TTSA	140s	140s	-
ASAG	0,41	0,47	14%

a prediction error. The precision for the TTSA metrics can be improved by reducing the batch size for the calculation of mean response time. The precision of the SLR is limited due to the load generator for the performance prototype that cannot reliably generate precise requests rates. The prediction error for the ASAG metric is 14% for our Znn.com evaluation example.

Figure 5.12 shows a comparison of the two time series from our evaluation. The orange-colored time series shows the predicted response time of our Znn.com performance model for the high-load usage context (first 140 seconds). The red-colored, dashed time series shows the response time measurements from our performance prototype in the same high-load usage context. A dashed blue, horizontal line marks the hard upper threshold ut_{hard} for our mean response time SLO.

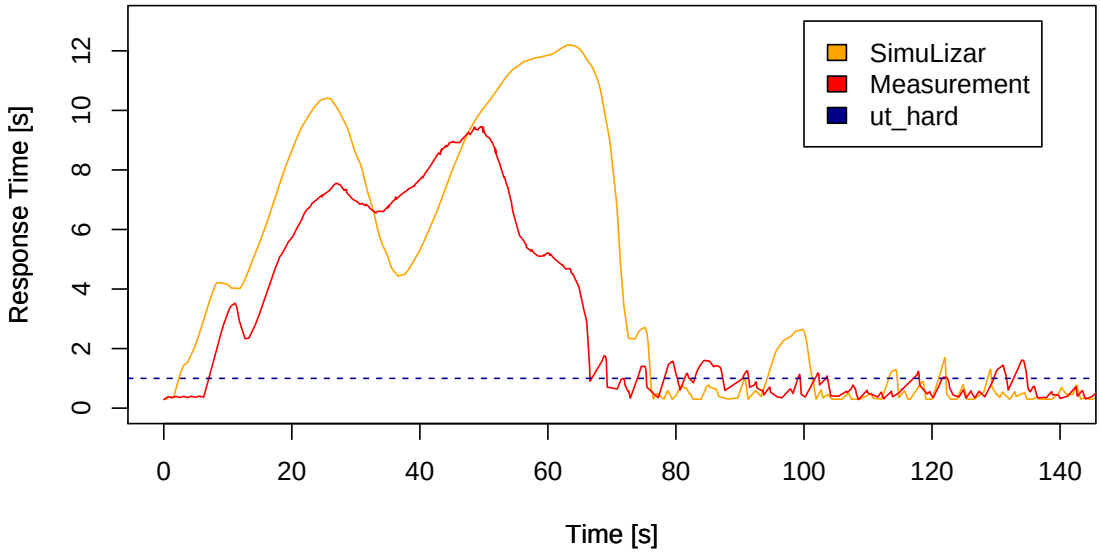


Figure 5.12.: Comparison of the predictions and measurements in a time series.

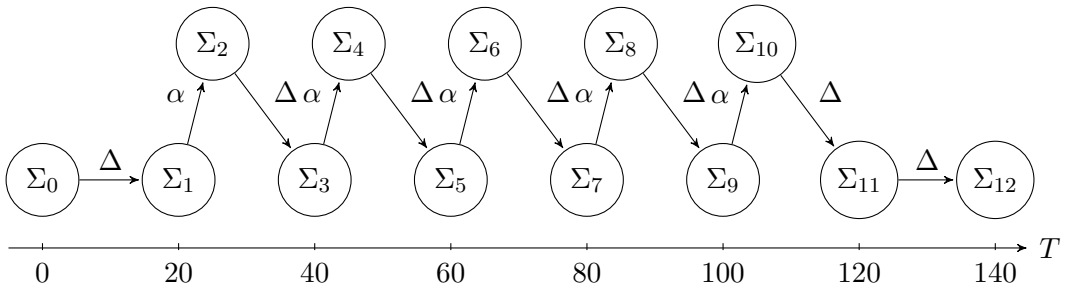


Figure 5.13.: State trace of the Znn.com system.

Table 5.5.: Values of the Znn.com state trace

Σ	Σ_0	Σ_1	Σ_2	Σ_3	Σ_4	Σ_5	Σ_6	Σ_7	Σ_8	Σ_9	Σ_{10}	Σ_{11}	Σ_{12}
a	a_0	a_0	a_1	a_1	a_2	a_2	a_3	a_3	a_4	a_4	a_5	a_5	a_5
m	$undef$	2.9	2.9	7.0	7.0	7.9	7.9	2.6	2.6	1.0	1.0	0.7	0.8
s	s_0	s_0	s_0	s_0	s_0	s_0	s_0	s_0	s_0	s_0	s_0	s_0	s_0
t	0	20.0	20.1	40.0	40.1	60.0	60.1	80.0	80.1	100.0	100.1	120.0	140.0

The self-adaptive system trace in Figure 5.13 corresponds to the measurements from the performance prototype in Figure 5.12. The trace shows each self-adaptive system state and the state transitions of the performance prototype during its execution. The state transitions are labeled with the function that triggered the state change. In the figure, the states in the lower row are triggered by new measurements, i. e., new values for the monitoring vector m from function Δ. The states in the upper row in the figure are triggered by reconfigurations, i. e., new architecture configuration values a from function α. The context scenario s does not change in the evaluated example.

The concrete values for the architecture configuration a, measurement vector m, and context scenario s for each state of the performance prototype are listed in Table 5.5. Note that we rounded the values for the mean response times in the measurement vector m as well as in the time t to one digit after the decimal mark.

Figure 5.14 shows a box plot for our evaluation. Each box shows all response times in 20 second batches for the predictions in orange boxes and the measurements in red boxes. Thus, each box corresponds to one architecture configuration a in the upper row of Figure 5.13.

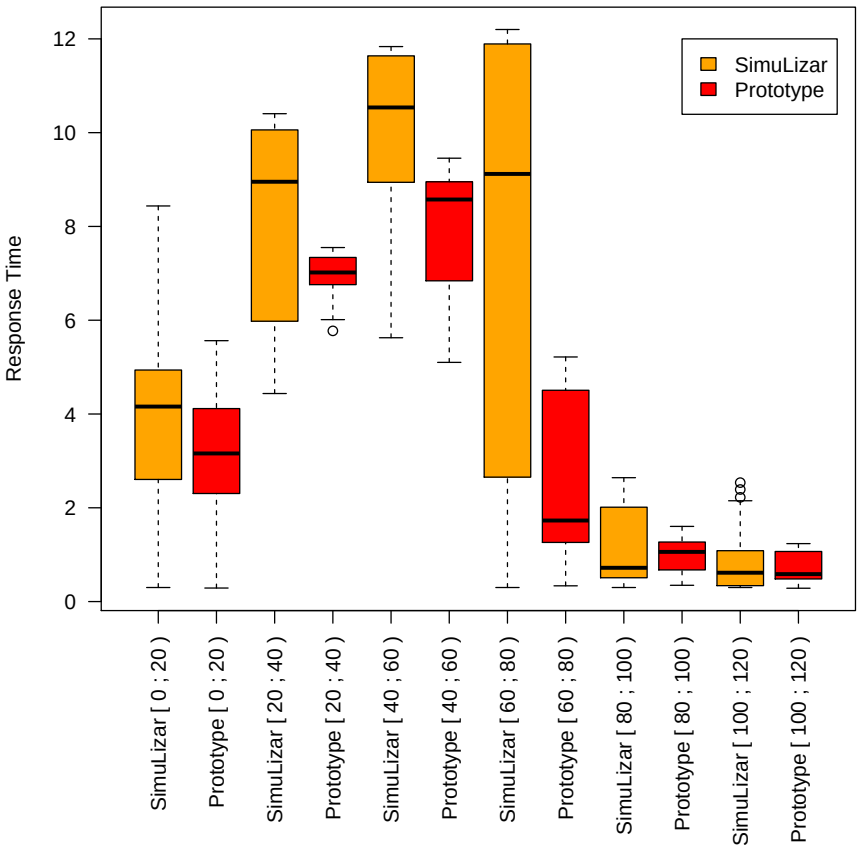


Figure 5.14.: Comparison of the predictions and measurements in a box plot.

5.9.4. Discussion

Our case study results show that our hypothesis H(applicable) holds and we can answer our evaluation question Q(applicability) with "yes".

According to Table 5.4, the prediction error for our ASAG metric is 14%. The prediction error for the other metrics could not be calculated due to insufficient precision. However, we estimate the metric for our case study i. e., M(prediction error) = 14%. Consequently, we can assume that our hypothesis H(applicable) holds, since the prediction error is likely to be less than 30%.

We can support the positive evaluation by inspecting the results of our evaluation in some more detail. What we can observe in the time series in Figure 5.12 as well as in

the box plot in Figure 5.14 is that the prediction accuracy for the steady states of the self-adaptive system are not very accurate, e. g., in the time interval between 20 seconds and 80 seconds. In this interval, the time series of the prediction and the measurements share some common characteristics, such as local minima and local maxima, but the distance of both curves points at a low prediction accuracy. The same can be observed in the box plot, where the three box pairs for the intervals $[20; 40)$, $[40; 60)$, and $[60; 80)$ also show large deviations.

With respect to the properties elasticity and scalability, however, the time series as well as the box plot indicate a sufficient prediction accuracy. First, our ASAG metric shows that predicted accumulated SLO achievement grade is close to the measured accumulated SLO achievement grade, with a prediction error of only 14%. This indicates that the distance between the two curves is not that drastic in terms of the prediction of the SLO achievement. Second, we can observe in the time series that the predicted as well as the measured response times drop below the hard upper threshold at about the same time. This is also reflected by our TTSA metric.

The results of our case study also indicate that our prediction methods fulfill our prediction methods requirements PR1 to PR7. The implementation of both prediction methods integrates model-driven software performance prediction tools for design-time prediction. We used the same self-adaptive system performance model for our prediction that we describe in the evaluation of our modeling approach in Chapter 4. Thus, the general requirements PR1 (“Design-Time”), PR2 (“Model-Driven”) are fulfilled. Requirement PR3 (“Integrated Tool Chain”) is fulfilled as well, since we integrated the implementation of our prediction methods in the same tool as the modeling approach. The implementation is described in detail in Chapter 6. As required by PR4 (“Formally Defined Metrics”), we formally defined metrics for scalability and elasticity based on our formalization with FBTL formula. Furthermore, we provided pseudo code implementations for both of our prediction methods, as required by PR5 (“Prediction Method”). Finally, both prediction methods take the system configuration and system context of a self-adaptive system into account, as both are part of the input for both prediction methods. Thus, also the requirements PR6 (“System Configuration”) and PR7 (“System Context”) are fulfilled.

In summary, our evaluation indicates that our prediction methods are applicable in order to predict the scalability and elasticity of self-adaptive systems. Furthermore, all of our prediction method requirements are fulfilled. Thus, we can conclude that the presented prediction methods achieve our goal to enable the prediction of scalability and elasticity at design-time.

5.9.5. Threats to Validity

In this section, we discuss the threats to the validity of the case study results and the conclusions that we have drawn from these results.

We can identify two threats to the validity of our case study and conclusions. First, the *selection of the case* for the case study poses a threat to the validity of the conclusions we have drawn from the results. Second, the *case study setup* poses a risk to the validity of the results of the case study.

Selection of the Case

The self-adaptive system that we selected for our case study, the Znn.com system, is a synthetic system from academia and no real system from industry. Thus, it could potentially be the case that self-adaptive systems in industry are designed and implemented essentially different from the Znn.com system. However, we assessed the risk of this threat as low, since the Znn.com system has been used as a benchmark system for the class of self-adaptive systems in various research evaluations [GCH+04, CGS09, CG12, CdL12, AAIW16]. In general, it is possible that the answers to our evaluation question would be different if the case study were applied to a different system. Due to the lack of well-documented and available self-adaptive systems the evaluation was limited to the Znn.com system. Still, we see the need for more evaluations, especially in an industrial context. For this purpose, our case study design can be reused also in industrial case studies.

Case Study Setup

The evaluation was conducted based on a performance prototype. This approach has the advantage that potential inaccuracies of the model do not threaten the complete evaluation. However, since we had to manually adapt the performance prototype, there is the threat that our implementation is incorrect or at least does not reflect the self-adaptation model. To counter this threat, we conducted a code review of the implementation, in which no major implementation errors could be found. An alternative case study setup would have been to take measurements from a real implementation of the Znn.com system instead of the performance prototype. However, a threat to the validity in this alternative setup would have been that the self-adaptive system performance model that was used to predict scalability and elasticity does not accurately reflect the implementation. Thus, we decided in favor of the presented case study setup.

In summary, we see only few threats to the validity with low risks for our evaluation results. However, since the case study was conducted in an academic context, we recommend repeating the evaluation with measurements from a real self-adaptive system in an industrial context.

5.10. Conclusion

In this chapter, we presented prediction method requirements that specify the general properties for model-driven prediction methods and specific properties for scalability and elasticity prediction methods. We discussed existing model-driven performance prediction methods according to these requirements and found that none of the discussed methods fulfills all requirements. We identified Palladio as the best basis for our prediction methods. However, Palladio is still lacking formally defined metrics, a scalability and elasticity prediction method, and means to analyze the complete reconfiguration space of a self-adaptive system. Hence, we presented a formalization of self-adaptive systems and service level objectives to also formally define scalability and elasticity metrics. Furthermore, we described prediction methods how to obtain our scalability and elasticity metrics based on Palladio.

The presented prediction methods enable the assessment of self-adaptive system architectures at design-time and scalability and elasticity properties. Thus, scalability and elasticity can be assured early in the software engineering process and project failure due to scalability and elasticity issues can be averted.

We evaluated the applicability of the presented prediction methods within a case study. The case study showed that our prediction methods are applicable to the Znn.com system. However, further empirical studies have to be conducted in order to show the applicability of the prediction methods in industrial software engineering projects.

"All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change."

Grady Booch

6

Tool Support

Contents

6.1. Architecture Overview 173

6.2. User Interface 176

6.3. Modeling Viewpoints 177

6.4. Performance Analysis Tool 180

6.5. Scalability and Elasticity Prediction 186

In this chapter, we present SimuLizar Bench, the implementation of our model-driven performance analysis method for self-adaptive systems. SimuLizar Bench implements both, our modeling approach that we presented in Chapter 4 as well as our prediction methods that we presented in Chapter 5.

The scalability and elasticity prediction methods that we presented in the previous chapter are based on our formalization of self-adaptive systems and service level objectives. For both methods, we provided metrics and pseudo code implementations for both properties. Our pseudo code implementations refer to two specific functions that are required to obtain these metrics. First, the function $\Delta(p_i, a, s, t)$ that determines monitoring values for the monitored property p_i at time t in a self-adaptive system. Second, the function to determine the grade an SLO is achieved in a given state of a self-adaptive system, i. e., $\Sigma_a \models Q_{p_i}$. The first function, $\Delta(p_i, a, s, t)$ is implemented by software performance analysis tools, like Palladio [BKR09] or LQNS [FAOW+09], for non-adaptive systems. An implementation for self-adaptive systems, however, does not exist yet. Since the second function, $\Sigma_a \models Q_{p_i}$, is based on the first function, an implementation does not exist yet as well. However, in order to evaluate the prediction methods and apply them in practice, an implementation of both functions is required.

With SimuLizar Bench, we provide a performance analysis tool that realizes the function $\Delta(p_i, t)$ not only for non-adaptive but also for self-adaptive systems. Furthermore, SimuLizar Bench implements the function $\Sigma_a \models Q_{p_i}$, to determine the grade an SLO is achieved in a given state of a self-adaptive system. Additionally, SimuLizar Bench provides broader tool support for modeling self-adaptive systems as described in Chapter 4.

Thus, with SimuLizar Bench, software engineers have a tool to model and evaluate self-adaptive system architectures. SimuLizar Bench supports the modeling process, we presented in Section 4.4, and integrates our scalability and elasticity prediction methods.

In this chapter, we first give an overview of SimuLizar Bench's software architecture in Section 6.1. Second, in Section 6.2, we briefly introduce the user interface in SimuLizar Bench. Third, the implementation of the modeling viewpoints, we presented in Chapter 4, is described in Section 6.3. Fourth, we present our model-driven performance analysis tool for self-adaptive systems in Section 6.4. Finally, in Section 6.5, we de-

scribe how service level objectives are evaluated for self-adaptive system performance models at design-time and how our scalability and elasticity metrics are obtained.

6.1. Architecture Overview

In this section, we provide an overview of the main components and dependencies of SimuLizar Bench. An initial implementation of SimuLizar Bench is described by Meyer [Mey11] in detail. In this chapter, we focus on the most relevant concepts and implementation details.

6.1.1. Main Components

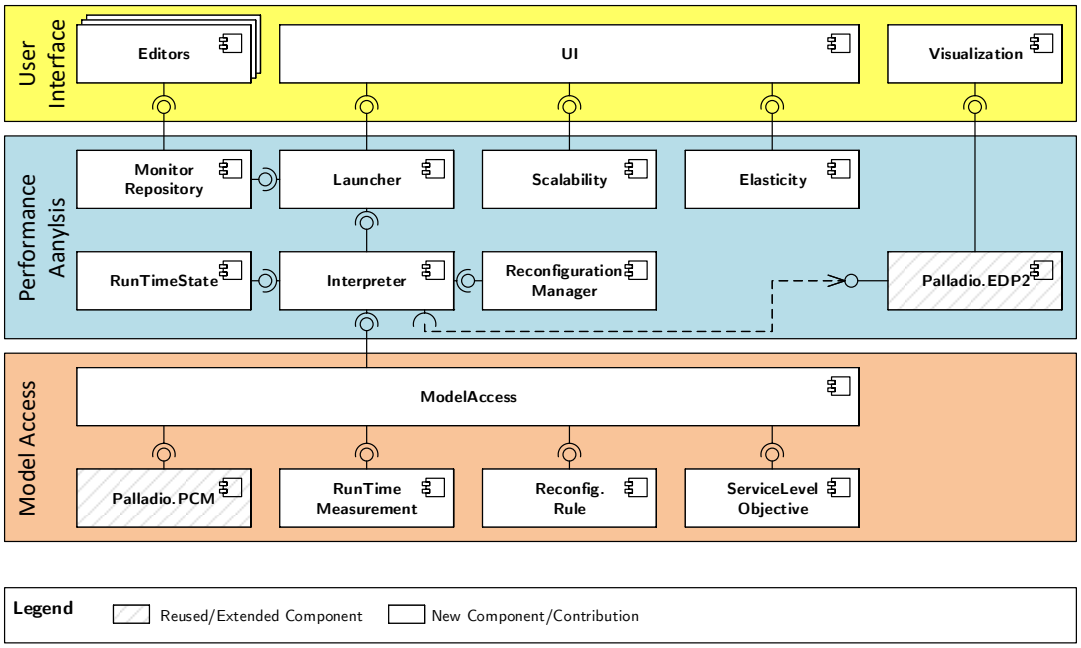


Figure 6.1.: Component diagram showing the SimuLizar architecture.

Figure 6.1 depicts a UML component diagram showing a high-level architecture and the most important components in SimuLizar Bench. The high-level architecture shows three layers: a user interface layer, a performance analysis layer, and a model access layer.

In the user interface layer, all user-facing functions such as the main user interface perspective and configuration dialogs are realized. Furthermore, modeling viewpoints with editors for SLOs, the monitor repository, and reconfigurations as well as the new visualizations for performance analysis results are implemented as part of the user interface layer. We briefly describe the implementation of the modeling viewpoints in Section 6.3. The new visualizations are described together with our prediction methods.

The performance analysis layer implements a performance analysis and forms the basis for our prediction methods that we describe in Section 6.4. We describe the implementations of the prediction methods in Section 6.5 where we also describe the new performance analysis result visualizations.

Finally, the model access layer provides access to the complete self-adaptive system performance model and a run-time measurements model that is used in the performance simulation. The models are described in detail in Chapter 4. Apart from the implementation of the models, the model layer only contains a model access component, which serves as access facade for the performance analysis layer components.

6.1.2. External Dependencies

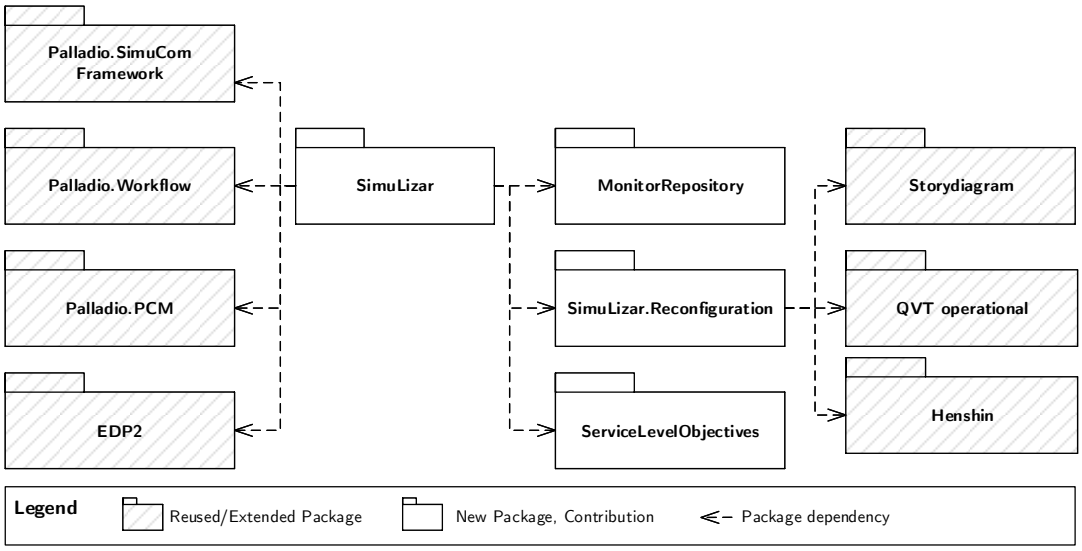


Figure 6.2.: Overview of packages and dependencies.

Figure 6.2 depicts a UML package diagram showing the SimuLizar package and its dependencies to external software packages. SimuLizar depends on four different Palladio components like depicted on the left side of the diagram: the *workflow engine*, the *PCM*, the *SimuCom framework* [Bec08], and the *Experiment Data Persistence & Presentation (EDP2)*. The first component, the workflow engine, is used to execute all necessary steps for our prediction methods, such as reading the input models, running performance analyses, and evaluating the SLO achievement. The PCM, the second Palladio component, builds the basis for our self-adaptive system performance model. Furthermore, the PCM editors are used to model single views of the model, as described in Chapter 4. The third component, SimuCom, provides the basis for the performance simulation. It is used to simulate requests, resource scheduling, and resource consumption for a performance model. Finally, EDP2 is used to persist measurements that are taken during the performance simulation. EDP2 also provides a framework for the visualization of the persisted measurements.

With SimuLizar Bench, we provide two software packages that can be used independently of SimuLizar's prediction methods. First, our *service level objectives* provide editors and wizards to create service level objectives, like described in Section 5.6. Second, with the *monitor repository*, we provide an editor to model performance monitors for SimuLizar models as well as PCM models, as described in Section 4.7.2.

Finally, SimuLizar provides a model editor to specify reconfigurations, as described in Section 4.7.3. A reconfiguration model can include preconditions and actions that are specified in three different model transformation languages: Storydiagram [FNTZ00], QVT operational [Obj16], and Henshin [ABJ+10]. All of these model-transformation languages are supported by our performance analysis for self-adaptive systems. However, while the elasticity prediction method in SimuLizar Bench supports all model-transformation languages, the scalability prediction method currently supports Storydiagram model transformations only.

6.2. User Interface

SimuLizar Bench provides tool support for modeling self-adaptive system architectures and predict these architecture's scalability and elasticity properties. SimuLizar Bench is implemented as an Eclipse [The16a] plug-in and is based on Palladio Bench [Kar16a]. The Palladio Bench provides modeling editors for the Palladio Component Model and performance prediction tools for non-adaptive systems, i. e., an implementation of the Palladio approach [Kar16b]. Palladio Bench also provides a workflow engine, frameworks for performance simulation, measurement persistence, and measurement visualization amongst others.

Figure 6.3.: Screenshot of SimuLizar Bench.

SimuLizar Bench reuses the PCM modeling editors from Palladio Bench and provides new editors for *monitor repositories*, *service level objectives*, and *reconfigurations*. Furthermore, SimuLizar Bench provides an interpreter-based performance analysis for self-adaptive system performance models, which is based on Palladio's SimuCom performance simulation framework. The performance-analysis is used in the implementation of our scalability and elasticity prediction methods. Finally, SimuLizar Bench also pro-

vides additional visualizations in Palladio Bench's result view for the inspection and manual analysis of performance simulations.

Figure 6.3 shows a screenshot of SimuLizar Bench. The screenshot shows the SimuLizar perspective with (1) the project browser on the left, an (2) initial system architecture configuration editor, the (4) results view with the new SLO achievement visualization, and (4) the console view with results from a performance analysis run.

In the following sections, we describe the architecture and implementation of SimuLizar Bench in more detail. In Section 6.1, we provide an overview of the SimuLizar Bench architecture and identify its main building blocks. Subsequently, we discuss the three main building block in the remaining section of this chapter. First, the implementation of our modeling viewpoints is presented in Section 6.3. Second, in Section 6.4, we present our interpreter-based performance analysis that build the basis for our scalability and elasticity prediction methods. Finally, we present the implementation of our prediction methods in Section 6.5.

6.3. Modeling Viewpoints

In SimuLizar Bench we implement the system type viewpoint and run-time viewpoint as proposed by Becker [Bec11] and added a third self-adaptation viewpoint. For all views types in these three viewpoints, which we presented in Chapter 4, we provide modeling editors, which are integrated in SimuLizar Bench. In order to support our modeling process, as presented in Section 4.4, we split the self-adaptive system performance model according to the view types. Thus, each view type is realized as a partial model, i. e., it contributes to the complete self-adaptive system performance model, but within dedicated model artifacts.

In the following sections, we briefly outline which model artifacts are part of the two self-adaptive system performance model viewpoints.

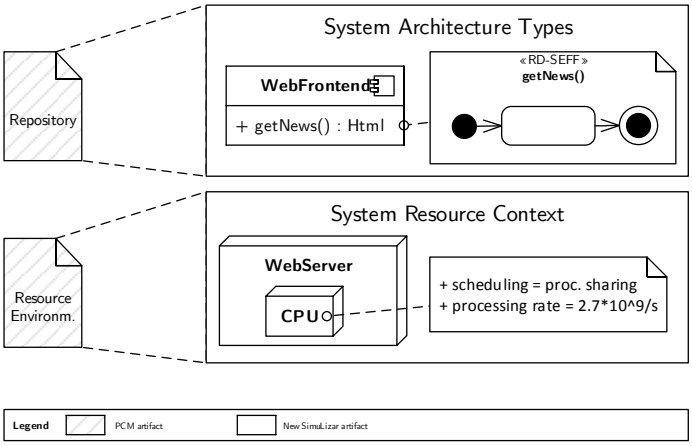


Figure 6.4.: System Type Viewpoint and its partial models in SimuLizar.

6.3.1. System Type Viewpoint

The system type viewpoint in SimuLizar Bench provides two modeling editors for the two type-level artifacts in a self-adaptive system performance model. In Figure 6.4 these two artifacts are illustrated. First, component developers model system architecture types with the PCM repository model editor. Second, in parallel to modeling the system architecture types, platform providers model the offered system resource contexts, i. e., cloud platforms like IaaS or PaaS environments.

6.3.2. Run-Time Viewpoint

The run-time viewpoint in SimuLizar Bench contains three modeling editors for the three self-adaptive system performance model artifacts in this viewpoint. The three artifacts are illustrated in Figure 6.5. First, the repository model files that are created by component developers is used by a self-adaptive system architect to specify an initial system architecture configuration with the PCM system model editor. Second, an initial system deployment is specified by an platform expert with the PCM allocation model editor. The PCM allocation model links to a resource environment model that was created by a platform provider. Finally, a domain expert specifies the system usage context with PCM's usage model editor.

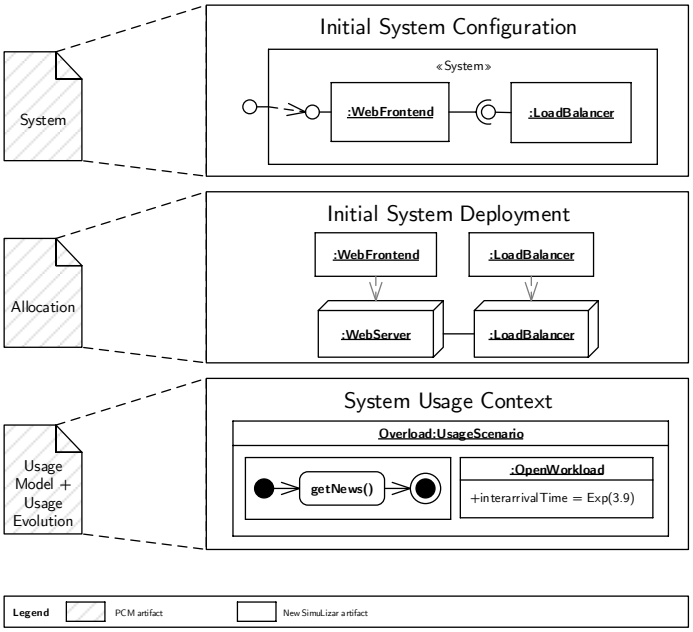


Figure 6.5.: Run-Time Viewpoint and its partial models in SimuLizar.

6.3.3. Self-Adaptation Viewpoint

Finally, the self-adaptation viewpoint in SimuLizar Bench contains three modeling editors for the three model artifacts in that are concerned with self-adaptation. The artifacts in this viewpoint are illustrated in Figure 6.6.

First, a domain expert models service level objectives with SimuLizar's SLO model editor. Second, after an initial system architecture and deployment has been specified, a platform expert models a monitor repository with SimuLizar's monitor repository editor. The platform expert has to specify monitors for all relevant quality properties, such that the SLOs can be achieved autonomously by the self-adaptive system. The monitor repository serves as an input for a self-adaptive system architect to specify reconfigurations in the run-time viewpoint. Finally, the third partial model is the reconfiguration repository. This model is specified by a self-adaptive system architect with SimuLizar's reconfiguration repository model editor.

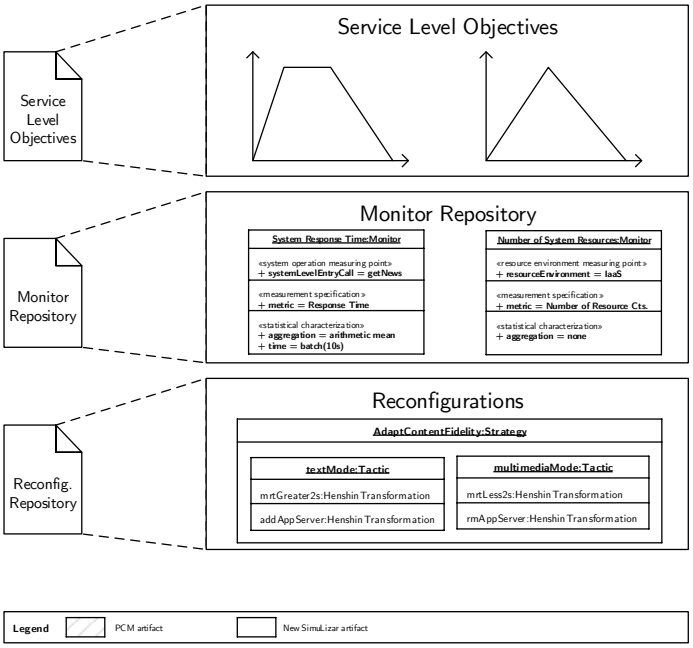
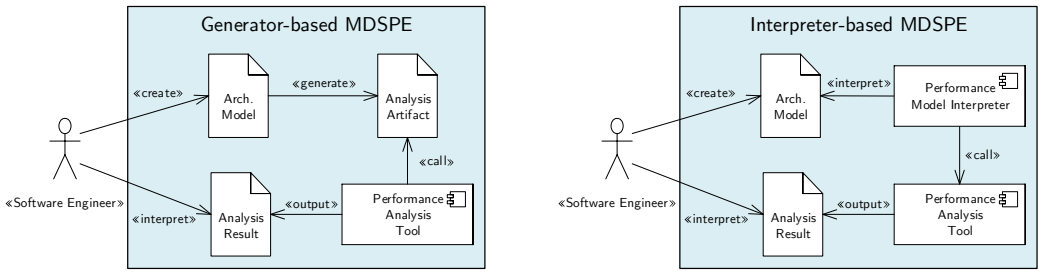


Figure 6.6.: Self-Adaptation Viewpoint and its partial models in SimuLizar.

The self-adaptive performance system model is complete when all partial models, as described above, have been specified. The complete model is used for the prediction of scalability and elasticity with SimuLizar's integrated self-adaptive system performance analysis tool.

6.4. Performance Analysis Tool

In SimuLizar Bench, we implement an interpreter-based performance analysis tool in contrast to generator-based performance analysis tools, which we introduced in Section 3.3. As illustrated in Figure 6.7, an interpreter-based performance analysis tool does not generate an analysis artifact, like simulation code, from an architecture model, but directly interprets the architecture model. For this purpose, a performance model interpreter continuously traverses the architecture model and calls a performance analysis tool, e. g., a performance simulation, whenever a model element with performance annotations, such as PCM's RD-SEFFS, is traversed.



(a) Generator-based performance simulation.

(b) Interpreter-based performance simulation.

Figure 6.7.: Comparison between generator-based and interpreter-based model-driven software performance engineering. [Mey11]

In general, both approaches, the generator-based approach and the interpreter-based approach, are equivalent with respect to their applicability to non-adaptive software system architectures. However, on the one hand, the generator-based approach may have slightly better performance, since the architecture model is only read once and native simulation code can be generated. On the other hand, the interpreter-based approach allows to continuously traverse the architecture model during the simulation. This approach has the advantage that we can adapt the architecture model while we simulate it. [Mey11] In contrast, a generator-based approach would require pausing the simulation, regenerating the simulation code, and then continuing the simulation. In SimuLizar Bench, we implemented an interpreter-based performance simulation to simulate the performance of a self-adaptive system and to predict scalability and elasticity properties.

As illustrated in Figure 6.8, our interpreter-based performance analysis tool in SimuLizar Bench consists of three components: (1) a *performance model interpreter*, (2) a *performance simulation* component, and (3) a *reconfiguration manager*. The additional reconfiguration manager in our interpreter-based performance analysis tool observes the analysis results, i. e., a run-time measurements model, and executes model-transformations that transform the architecture model. The model transformations executed by the reconfiguration manager are part of the reconfigurations in our self-adaptive system performance model. Thus, SimuLizar Bench's performance analysis

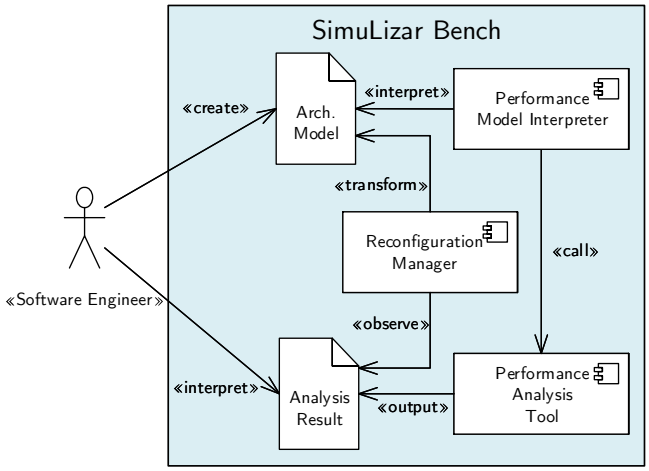


Figure 6.8.: Self-Adaptive System Performance Analysis in SimuLizar Bench.

tool does not only simulate a static architecture configuration but a self-adaptive system architecture with its architecture reconfigurations.

In the following subsections, we describe the three main components of our performance analysis tool. First, we describe the performance model interpreter in Section 6.4.1. Second, we describe the performance analysis tool in Section 6.4.2. Finally, in Section 6.4.3, we describe the reconfiguration manager.

6.4.1. Performance Model Interpreter

In SimuLizar Bench, a self-adaptive system performance model is continuously traversed for the interpretation and simulation. Like for other simulation-based performance analysis tools, like Palladio's SimuCom, in SimuLizar Bench each user request is simulated individually.

Figure 6.9 depicts a sequence diagram of an exemplary traversal of a self-adaptive system performance model for one user request. As shown in the sequence diagram the starting point of our self-adaptive system model interpreter is the usage context. The usage context provides information about the load and work to be simulated, as described in Section 4.6.3. The work specification in the usage context specifies how a

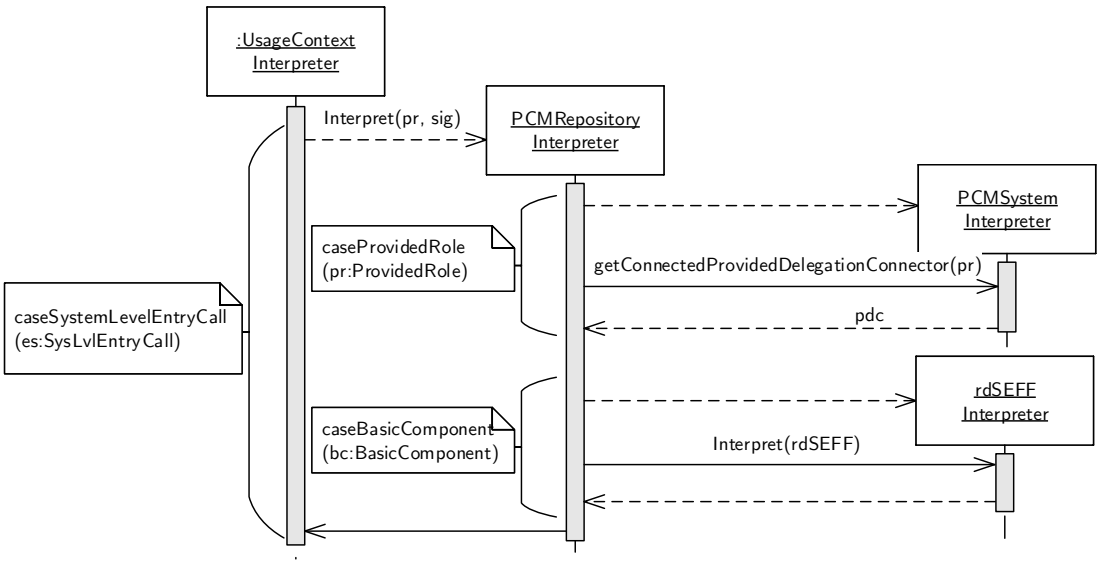


Figure 6.9.: Interpretation of a performance model. [Mey11]

single user behaves and sends requests, i. e., the user calls SystemLevelEntryCalls, to the self-adaptive system. The load specification specifies at which rate the users execute their work, i. e., it defines the request rate.

The behavior of users is interpreted by the UsageContext Interpreter, as depicted in the sequence diagram. Whenever the UsageContext Interpreter traverses an SystemLevelEntryCall element, it calls the PCMRepository Interpreter with the signature sig of the SystemLevelEntryCall and the system's provided role pr that provides the SystemLevelEntryCall. The PCMRepository Interpreter is responsible to interpret system architecture elements like interfaces, components and the roles of components. In the illustrated case, the PCMRepository Interpreter calls the PCMSystemInterpreter to get the component with the provided role pr. The PCMSystem Interpreter returns the basic component pdc. The basic component pdc is then interpreted by the PCMRepositoryInterpreter again. The PCMRepository Interpreter calls the rdSEFFInterpreter for the signature sig. Finally, the rdSEFFInterpreter calls the performance analysis tool, to simulate the performance behavior of the RD-SEFF, like described in Section 6.4.2.

A more detailed description and implementation details of the interpreter can be found in [Mey11].

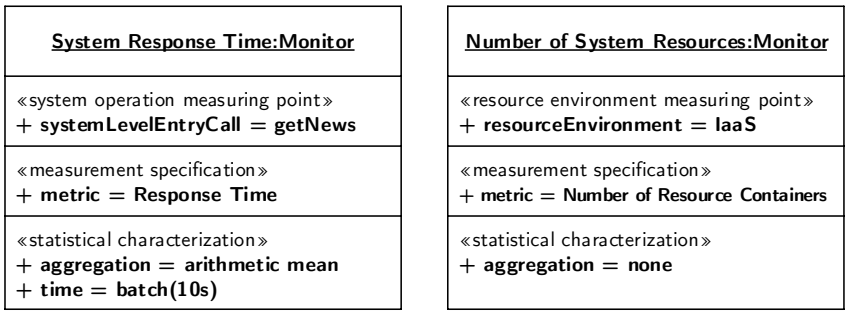


Figure 6.10.: Monitor repository for the Znn.com system.

6.4.2. Performance Simulation

The interpreter as described in the previous section simulates the propagation of user requests through the self-adaptive system. Whenever a resource demanding behavior, i. e., a RD-SEFF, is reached, Palladio's SimuCom framework is called to simulate the performance behavior. That is, the SimuCom framework takes care of the simulation of resource scheduling and the simulation of resource congestion, etc. A detailed description of the performance simulation with the SimuCom framework can be found in [Bec08].

The monitoring repository model specifies at which elements of the system performance measurements are taken and how these are aggregated. Measurements are then either taken event-triggered when a certain system element is simulated or periodically, e. g., every 10 seconds.

Figure 6.10 shows the monitor repository from our Znn.com example with two monitor specifications: System Response Time and Number of System Resources. The first monitor specifies an event-triggered monitor for a SystemLeveLEntryCall named getNews. At this measuring point, the metric Response Time shall be obtained and aggregated as an arithmetic mean in 10-second batches. That is every time the interpreter passes by the getNews model element, a measurement is taken. The second monitor specification, Number of System Resources, is a periodically triggered monitor. That is, at every time unit of the interpretation and simulation, a measurement is taken.

The raw measurements, i. e., non-aggregated measurements, are stored via Palladio's EDP2 persistence and visualization framework. After the simulation, the self-adaptive system architect can view the raw measurements in the result view, as shown in Figure 6.3. The result view supports different visualizations.

All measurements that are taken during the simulation of the self-adaptive system are also aggregated according to the monitor specifications and stored in a run-time measurement model. In the example above, all measurements for getNews that are taken within one 10-second batch are aggregated to a single mean response time value. Only this single, aggregated mean response time is then stored at the end of the 10 seconds (of the batch) in the run-time measurement model. The run-time measurement model is used as an input for the reconfiguration manager, which we describe in the following subsection.

6.4.3. Reconfiguration Manager

SimuLizar Bench implements a MAPE-K-like control loop in which a self-adaptive system performance model is interpreted, simulated, and reconfigured. Figure 6.11 shows a conceptual overview of our MAPE-K control loop implementation.

In SimuLizar Bench's MAPE-K control loop, the managed element is the *global self-adaptive system performance model*. This global model always reflects the current architecture of the self-adaptive system. It is initially the same performance model as designed with the modeling editors in SimuLizar Bench.

For each simulated user request, the global self-adaptive system performance model is copied to a *local self-adaptive system performance model*. Thus, each user request is simulated with a dedicated local copy. Consequently, reconfigurations do not interfere with running simulations.

During the simulation, run-time measurements are taken as specified in the monitor repository model. This reflects the monitor phase of the MAPE-K control loop. In the analyze phase, the reconfiguration manager component checks the conditions for all tactics in the reconfiguration model. In the plan phase, the reconfiguration manager

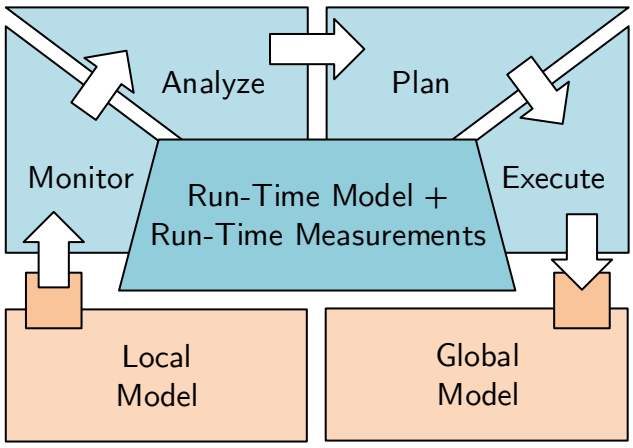


Figure 6.11.: MAPE-K feedback loop in SimuLizar.

selects the one tactic with fulfilled conditions and the highest priority. If multiple tactics with the same priority and fulfilled conditions are available, the reconfiguration manager selects the first of these tactics (according to the order in the model file). In the execute phase, the action of the selected tactic is executed.

We use model transformations to simulate reconfigurations. That is, a copy of the global self-adaptive system performance model is created and the model transformation of the action element in the reconfiguration model is executed on this copy. The transformed model then becomes the new global self-adaptive system performance model. Consequently, user requests that are simulated after the reconfiguration use a local copy of the transformed, i. e., reconfigured, global self-adaptive system performance model. Each user request simulation is completely performed using a single local self-adaptive system performance model, i. e., one architecture configuration.

6.5. Scalability and Elasticity Prediction

In this section, we describe our implementation of the scalability and elasticity prediction methods. Both prediction methods rely on the simulation-based performance analysis tool that we described in the previous section. With this tool, we can determine measurements for each monitored property at each point in time of the simulation,

i. e., function $\Delta(p_i, a, s, t)$. This function also allows us to predict the SLO achievement of a self-adaptive system at each point in time of the simulation. Thus, we can also predict the scalability and elasticity of the system, like described in Section 5.8.3 and Section 5.7.3.

In the following subsections, we describe how scalability and elasticity are predicted with SimuLizar Bench, how the prediction results are visualized, and how the results can be interpreted.

6.5.1. Scalability Prediction

For the prediction of scalability, we implemented our scalability prediction method that we described in Section 5.7.3. Specifically, our implementation uses a reachability analysis framework [Hei15] that provides convenient methods to explore the reconfiguration space with all architecture configurations that are reachable via the reconfiguration actions of the self-adaptive system performance model.

The scalability prediction is realized in two steps. In the first step, all reachable architecture configurations are created via the according model transformations. Initially, a hash value for the initial system architecture configuration model calculated. Next, the model transformations of each reconfiguration action are applied to the initial system architecture configuration. Again, a hash value for each of the resulting architecture configurations is calculated. Architecture configurations with equal hash values are dismissed. For all other unique architecture configurations, the model transformations of the reconfiguration actions are applied again. This process continues until no new, unique architecture configurations can be found. In the second step, the context scenario, specified in the run configuration, is simulated for each architecture configuration. Reconfigurations are deactivated in the simulation. The achievement of each SLO is calculated during the simulation. If an architecture configuration achieves all SLOs during the whole simulation time, the scalability prediction is stopped in advance and the user is notified that the simulated system scales up to the specified usage context. If no architecture configuration achieves all SLOs during the whole simulation time, the

scalability analysis notifies the user that the system does not scale up to the specified usage context.

The scalability metrics *scalability load range* and *scalability work range* can be determined by increasing the load or work step-wise for the specified usage context until the scalability prediction result is negative. This can be automated with Palladio's experiment automation.

6.5.2. Elasticity Prediction

The elasticity prediction in SimuLizar Bench implements our elasticity prediction method that we presented in Section 5.8.3. Our implementation uses Palladio's confidence stop condition that stops a simulation after a specified confidence level has been achieved. In Palladio usually a certain confidence level, e. g., a 95% confidence interval, for the mean response time metric of a usage context scenario shall be reached. In SimuLizar, however, we are not interested in the confidence level of the response time metric, but the confidence level of the elasticity metrics, i. e., *TTSA* or *ASAG*.

The elasticity prediction in SimuLizar Bench simulates a self-adaptive system performance model and obtains the elasticity metrics for the simulation run. SimuLizar Bench repeats the simulation run of the same model and obtains the elasticity metrics until the specified confidence level for the elasticity metrics is reached. For this purpose, the confidence level is calculated at the end of each simulation run by taking into account the elasticity metric values of all previous simulation runs. The result of the elasticity prediction in SimuLizar Bench are mean values for our elasticity metrics.

6.5.3. Prediction Result Visualization

We implement two new prediction result visualizations in SimuLizar Bench. Both visualizations help software engineers to identify whether the modeled self-adaptive system architecture achieves the specified SLO in the modeled context scenario.

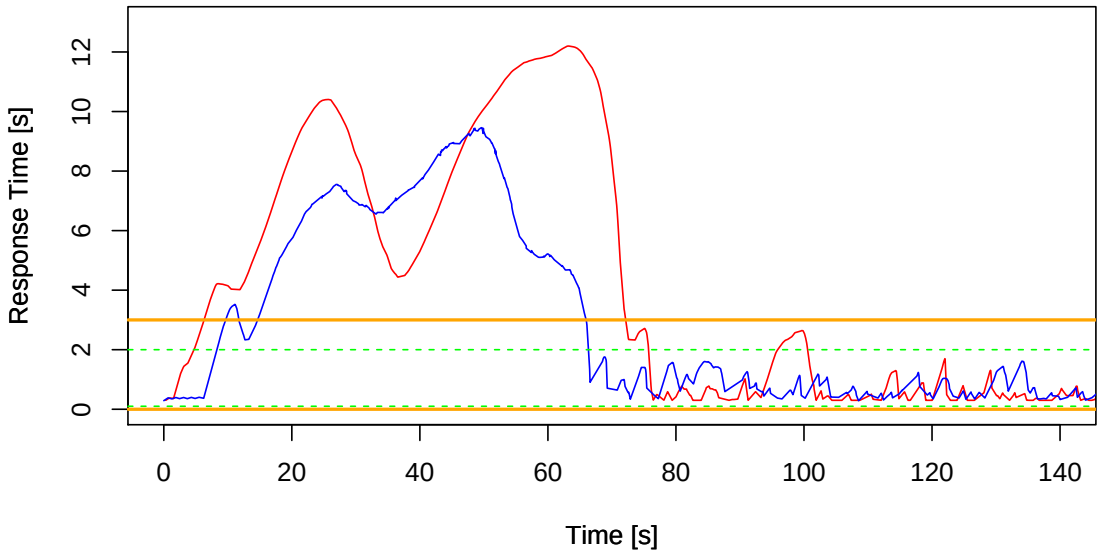


Figure 6.12.: SLO thresholds visualization in SimuLizar.

Figure 6.12 shows the first new prediction result visualization. The visualization shows a time series for the selected monitored element. The horizontal axis represents the simulation time, the vertical axis represents the measurement values. For example, if the user selects a response time monitor, each response time measurement in the simulation is shown as a point in the time series. Additionally, the user can select and SLO for which the soft thresholds and hard thresholds can be shown in a time series graph as horizontal lines. Thus, the user can quickly identify whether measurements were taken that exceeded the specified thresholds.

In the example, illustrated in Figure 6.12, the response time time series for a simulation run of our Znn.com system is shown. The horizontal axis represents the simulation time, the vertical axis represents the predicted response times for our Znn.com system in an overload context scenario. The hard thresholds of our SLO Q_{mrt} are shown as thick horizontal lines in the time series. The soft thresholds are shown as thin horizontal lines in the time series. In our example, we defined that the lower hard threshold and the lower soft threshold are both zero seconds, i. e., $lt_{soft} = lt_{hard} = 0.0$. The upper soft threshold is 2.0 seconds and the upper hard threshold is 3.0 seconds.

Figure 6.13 shows the SLO achievement grade visualization, i. e., our second prediction result visualization. In this visualization, the grade of SLO achievement is shown for

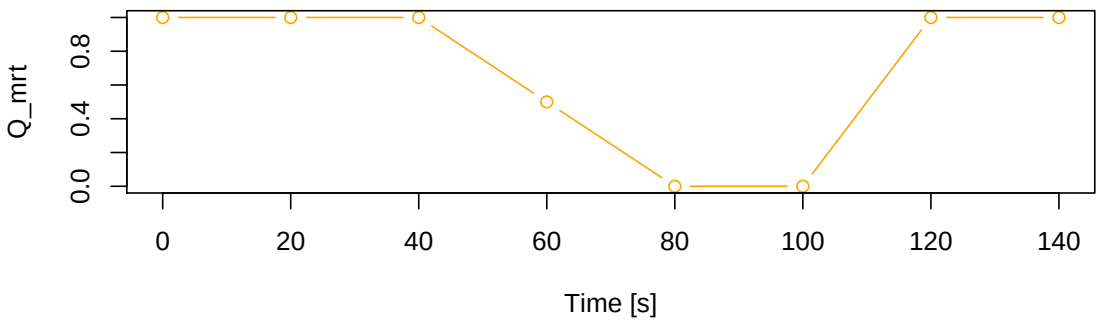


Figure 6.13.: SLO achievement visualization in SimuLizar.

each point in time. The horizontal axis represent the simulation time, the vertical axis represents the SLO achievement grade. For example, if the user selects a response time monitor and an according SLO, the graph shows the grade (between 0 and 1) to which the selected SLO is achieved at each time of the simulation. Thus, the user can quickly identify whether SLOs are fully achieved, partly achieved, or not achieved at all.

In the example, illustrated in Figure 6.13, a graph of the graded achievement of the SLO Q_{mrt} for our Znn.com example is shown. The horizontal axis represents the simulation time, the vertical axis represents the predicted SLO achievement for SLO Q_{mrt} in an overload context scenario. The graph shows, the SLO Q_{mrt} is completely achieved at the beginning of the simulation, but degrades after simulation time 40.0 until the SLO achievement reaches it lowest point at simulation time 80.0. After simulation time 100.0, the SLO achievement increases again until it reaches complete SLO achievement at time 120.0.

7

Conclusion

Contents

7.1. Results and Conclusions 192
7.2. Follow-Up Work 193
7.3. Future Work 195

In this final chapter, we first summarize and conclude the results of this thesis. Second, we provide a brief overview of follow-up work that is based on initial results of this thesis. Finally, we point to directions for future research challenges that continue with our research direction.

7.1. Results and Conclusions

In this thesis, we addressed the specification of self-adaptive systems and the assessment of the quality properties scalability and elasticity at design-time. We introduced SimuLizar, a model-driven performance engineering method that supports software engineers to design scalable and elastic software systems. SimuLizar Bench implements a tool chain for our performance engineering method. Thus, with SimuLizar Bench we provide software engineers with a tool to model self-adaptive systems and identify scalability and elasticity issues via simulation already at design-time.

The evaluation of our performance engineering method provides evidence that SimuLizar helps to identify design flaws in the self-adaptation layer. Consequently, project delays and project failures caused by unfulfilled scalability and elasticity requirements can be averted.

In the following, we briefly summarize our conclusions that we draw with respect to the problems in engineering self-adaptive systems and the solutions that we have presented in this thesis.

Self-Adaptive System Performance Modeling In this thesis we introduced a performance modeling approach for self-adaptive systems. For this modeling approach, we described a modeling process that also specifies roles, modeling tasks, and modeling artifacts.

The self-adaptive system performance model is organized into three viewpoints: a system type viewpoint, a run-time viewpoint, and a self-adaptation viewpoint. The system type viewpoint supports the specification of type elements like components and interfaces. The type level elements are instantiated and an initial system architecture

configuration and deployment is specified in the run-time viewpoint. Finally, in the self-adaptation viewpoint service level objectives, monitors, and reconfigurations are specified. Thus, the self-adaptation viewpoint reflects the self-adaptation layer.

We implemented our performance modeling approach in SimuLizar Bench and evaluated the approach in a case study. The evaluation results showed that our performance modeling approach is applicable to model self-adaptive systems. With our performance modeling approach, we provide the necessary precondition for the assessment of performance properties of self-adaptive systems at design-time.

Metrics for Scalability and Elasticity We formally defined self-adaptive systems and their properties scalability and elasticity. The formalization is based on the Fuzzy Branching Temporal Logic, which allowed us to define a notion of graded service level objective achievement. Service level objectives can be precisely defined and thus the level of imprecision that is inherent to requirements in natural language can be reduced.

Based on our formalization, we formally defined concrete metrics for scalability as well as elasticity. These metrics capture the quality of the self-adaptation layer and thus enable software engineers to compare alternative designs of this layer to each other.

Prediction of Scalability and Elasticity Based on our formalization and on our metric definitions, we provided prediction methods for our scalability and elasticity metrics. We implemented both prediction methods as part of SimuLizar Bench.

We evaluated our implementation of the prediction methods using the Znn.com system. The evaluation showed that our scalability and elasticity predictions provide sufficient prediction accuracy to compare alternative system designs and detect design flaws in the self-adaptation layer early at design-time.

7.2. Follow-Up Work

Based on initial results in the context of this thesis two follow-up research projects have been started. In both follow-up projects SimuLizar is used for predictions of

non-functional properties in the context of cloud-based, self-adaptive systems. In the following paragraphs, we briefly outline these projects and their use of SimuLizar.

CloudScale and Architectural Templates The CloudScale project [LB15] was an project funded by the European Commission under the Seventh Framework Programme. The project's goal was to support software engineers to predict and resolve scalability issues in cloud-based services. To achieve this goal, two methods were developed.

First, a re-engineering method was developed that helps to refactor legacy software and migrate the software to cloud platforms. Second, with the *Architectural Template Method* [Leh14], an engineering method for analyzing quality properties of software architectures at design-time. For this purpose, architectural knowledge, e. g., architecture styles and architecture patterns, is provided in form of reusable templates, called *Architectural Templates* (AT). An AT also provides, besides structure and behavior specifications, parametric performance annotations that can be used for performance predictions.

Consequently, software engineers can easily apply ATs to model software systems and predict performance properties. Thus, the effort for model-driven software performance engineering is reduced. The AT method also supports the specification of self-adaptive systems by providing some ATs that specify templates for self-adaptive systems. These templates are based on the self-adaptive system performance model that we presented in this thesis.

Cactos and CactoSIM The Cactos project [ÖGW+14] was an project funded by the European Commission under the Seventh Framework Programme as well. The goal of the Cactos project was to provide a full tool chain for monitoring, optimization, and simulation of cloud-based software systems.

In the Cactos project, three components were developed that build the tool chain: CactoScale as a monitoring solution, CactoOpt as an optimization solution for IaaS data centers, and CactoSim as a simulation tool. CactoSim [SK16] is based on SimuLizar and extends its modeling views with new annotations for energy consumption. Thus,

CactoSim can be used to predict scalability, elasticity, and energy consumption of cloud-based software systems.

7.3. Future Work

The follow-up work that is based on the results of this thesis shows some directions for future work. Additional to these research directions, we want to point to some more directions for future work in the context of this thesis.

Additional Metrics In this thesis, we provide formally defined metrics for scalability and elasticity. Both metrics are properties of self-adaptive systems that characterize the quality of the self-adaptation layer. However, other metrics can be defined that help to further characterize the self-adaptation layer, e. g., competitiveness and robustness.

Competitiveness is a metric to measure how good an algorithm performs in comparison to other algorithms. This metric could be used to compare the self-adaptation strategy to a theoretical optimal strategy. This optimal strategy has to be defined according the requirements.

Robustness is a desired property in control engineering. Self-adaptive systems, especially the MAPE-K control loop, works similar to a traditional control loop in control theory. Therefore, robustness is a property to consider in the context of self-adaptive systems as well.

A realization of more metrics would, first, require to formally define the metrics and, second, to extend our prediction method as well as our modeling approach if the metrics require more annotations within the model.

Reconfiguration Performance With our modeling approach it is currently only possible to model the resource demands of a complete reconfiguration rule, i. e., all four phases in the MAPE-K control loop together.

For more advanced analyses of the self-adaptation layer, a more detailed model of the resource demands for each phase, i. e., monitoring, analysis, planning, and execution,

will provide useful insights. To realize this, it is required to extend our self-adaptive system performance model such that tactics in our reconfiguration model contain elements that represent each of the reconfiguration phases. Additionally, it must be enabled to annotate a resource demand for each phase.

Uncertainty in Usage Contexts In Palladio as well as in SimuLizar, uncertainty in the usage context is modeled with probabilities, e. g., random variables for request rates. However, probability does only reflect aleatoric uncertainty, i. e., uncertainty that originates from an intrinsic randomness, and that can be modeled with random variables. In contrast, possibility reflects uncertainty that originates from the lack of knowledge and is modeled with possibility distributions. Since, it is impossible to know the future behavior of a human user, possibility distributions reflect the uncertainty in usage context better than random variables.

Consequently, an interesting direction for future work is to model the usage context in SimuLizar with possibility distributions instead of random variables. A realization would require to not only adapting the usage context meta model but also the simulation for the prediction methods to reflect the semantics of possibility distributions.

Design Space Exploration SimuLizar enables software engineers to assess the scalability and elasticity of self-adaptive systems. Still, finding the best parameters for the conditions and action of reconfigurations can be a time-consuming process. Hence, it is desirable to speed up this process, e. g., by automatic *design space exploration*.

With a design space exploration, Pareto-optimal parameters for the reconfiguration conditions and actions can be found automatically.

Evaluation We presented evaluations for our modeling approach as well as for our prediction methods in this thesis. Both evaluations can be extended in the context of an industrial field study. The applicability in an industrial context is a key factor of success for software engineering methods. Furthermore, a field study may reveal more insights in the current limitations the performance modeling approach, the prediction methods, and their implementation in SimuLizar Bench.

B

Bibliography

The bibliography is structured into four parts: own publications, supervised Bachelor's theses and Master's theses, literature cited in this thesis, and websites and standards referenced in this thesis.

Own Publications

[ABPW14] Svetlana Arifulina, Matthias Becker, Marie Christin Platenius, and Sven Walther. SeSAME: Modeling and Analyzing High-Quality Service Compositions. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, pages 839–842, New York, NY, USA, 2014. ACM. doi:10.1145/2642937.2648621.

[BBB+14] Matthias Becker, Steffen Becker, Galina Besova, Sven Walther, and Heike Wehrheim. Towards Systematic Configuration for Architecture Validation. In *Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (Work in Progress Session)*. IEEE, 2014.

[BBM13] Matthias Becker, Steffen Becker, and Joachim Meyer. SimuLizar: Design-Time modeling and Performance Analysis of Self-Adaptive Systems. In *Proceedings of the Software Engineering (SE)*, Lecture Notes in Informatics (LNI), pages 71–84. Gesellschaft für Informatik (GI), 2013.

[BLB12] Matthias Becker, Markus Luckey, and Steffen Becker. Model-driven Performance Engineering of Self-adaptive Systems: A Survey. In *Proceed-*

ings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), pages 117–122, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2304696.2304716, doi:10.1145/2304696.2304716.

[BLB13] Matthias Becker, Markus Luckey, and Steffen Becker. Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time. In *Proceedings of the 9th ACM SigSoft International Conference on Quality of Software Architectures (QoSA)*, pages 43–52, New York, NY, USA, 2013. ACM. doi:10.1145/2465478.2465489.

[BLB15] Matthias Becker, Sebastian Lehrig, and Steffen Becker. Systematically Deriving Quality Metrics for Cloud Computing Systems. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, volume 15, pages 169–174, New York, NY, USA, 2015. ACM. doi:10.1145/2668930.2688043.

[BPB14] Matthias Becker, Marie Christin Platenius, and Steffen Becker. Cloud Computing Reduces Uncertainties in Quality-of-Service Matching! In *Proceedings of the 2nd International Workshop on Cloud Service Brokerage (CSB)*, Communications in Computer and Information Science, pages 153–159, Berlin/Heidelberg, Germany, 2014. Springer. doi:10.1007/978-3-319-14886-1_15.

[KMFC+14] Alessia Knauss, Juan C. Muñoz-Fernández, Lorena Castañeda, Matthias Becker, Mahdi Derakhshanmanesh, Nina Taherimakhsousi, and Robert Heinrich. 2.3 Artifact-Centric Requirements Engineering for Self-Adaptive Systems. In *Report from the GI Dagstuhl Seminar 14433: Software Engineering for Self-Adaptive Systems*, page 7, 2014.

[LB14] Sebastian Lehrig and Matthias Becker. Approaching the Cloud: Using Palladio for Scalability, Elasticity, and Efficiency Analyses. In *Proceedings of the Symposium on Software Performance (SOSP)*, 2014.

[PSS+16] Marie Christin Platenius, Wilhelm Schäfer, Ammar Shaker, Eyke Hüllermeier, and Matthias Becker. Imprecise Matching of Requirements Specifi-

cations for Software Services using Fuzzy Logic. *Transactions on Software Engineering*, 2016. To appear. doi:10.1109/tse.2016.2632115.

Supervised Theses

[Bul14] Benjamin Bulk. *Evaluating the Influence of Different Abstraction Levels of Software Design on Performance prediction.* Master's thesis, Paderborn University, Paderborn, 2014.

[Flo15] Andreas Flohre. *Service Level Objective Fulfillment Reports für SimuLizar.* Bachelor's thesis, Paderborn University, Paderborn, 2015.

[Gop14] Vinay Akkasetty Gopal. *Dynamic Environment Model for Performance Analysis of Self-Adaptive Systems.* Master's thesis, Paderborn University, Paderborn, 2014.

[Hei14] Jonas Heinisch. *Migration der StochasticExpressions in Palladio mit Xtext.* Bachelor's thesis, Paderborn University, Paderborn, 2014.

[Joj14] Suman Jojiju. *Finding Optimal Self-Adaption Rules by Design-Space Exploration.* Master's thesis, Paderborn University, Paderborn, 2014.

[Moh12] Mario Mohr. *Generating Prototypes of Adaptive Component-based Software Systems for Performance Analysis.* Master's thesis, Paderborn University, Paderborn, 2012.

[Pis15] Goran Piskachev. *LAFORE: A Domain Specific Language for Reconfiguration.* Master's thesis, Paderborn University, Paderborn, 2015.

[Rog16] Igor Rogic. *Scalability and Elasticity Prediction of Self-Adaptive Systems Using SimuLizar and Architectural Templates: Industrial Case Study.* Master's thesis, Paderborn University, Paderborn, 2016.

Literature

[AAIW16] Nadeem Abbas, Jesper Andersson, Muhammad Usman Iftikhar, and Danny Weyns. Rigorous Architectural Reasoning for Self-Adaptive Software Systems. In *Proceedings of the 1st Workshop on Qualitative Reasoning about Software Architectures (QRASA)* [QRA16], pages 1–8. doi:10.1109/qrasa.2016.9.

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Petriu et al. [PRH10], pages 121–135. doi:10.1007/978-3-642-16145-2_9.

[ACC+14] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, JuanF Pérez, and Weikun Wang. Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 5(1):11–28, 2014. URL: http://dx.doi.org/10.1186/s13174-014-0011-3, doi:10.1186/s13174-014-0011-3.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010. URL: http://doi.acm.org/10.1145/1721654.1721672, doi:10.1145/1721654.1721672.

[AH96] Rajeev Alur and Thomas A. Henzinger, editors. *Proceedings of the 8th International Conference on Computer Aided Verification (CAV)*, Berlin/Heidelberg, Germany, 1996. Springer.

[BAB12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Computer Systems*, 28(5):755–768, 2012. doi:10.1016/j.future.2011.04.017.

[Bal09] Helmut Balzert. *Lehrbuch der Softwaretechnik Basiskonzepte und Requirements Engineering.* Spektrum Akademischer Verlag, Heidelberg, Germany, 3rd edition, 2009.

[Bal11] Helmut Balzert. *Lehrbuch der Softwaretechnik Entwurf, Implementierung, Installation und Betrieb.* Spektrum Akademischer Verlag, Heidelberg, Germany, 3rd edition, 2011.

[BCP12] Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors. *Formal Methods for Model-Driven Engineering.* Springer, Berlin/Heidelberg, Germany, 2012. doi:10.1007/978-3-642-30982-3.

[BDG+09] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering Self-Adaptive Systems through Feedback Loops. In Cheng et al. [CdLG+09a], pages 48–70. URL: http://dx.doi.org/10.1007/978-3-642-02161-9_3, doi:10.1007/978-3-642-02161-9_3.

[BDIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004. doi:10.1109/TSE.2004.9.

[Bec08] Steffen Becker. *Coupled Model Transformations Coupled Model Transformations for QoS Enabled Component-Based Software Design.* Dissertation, Universität Oldenburg, Oldenburg, 2008.

[Bec11] Steffen Becker. Towards System Viewpoints to Specify Adaptation Models at Runtime. In *Proceedings of the Software Engineering (SE)* [SE.11].

[BF08] Rainer Böhme and Felix C. Freiling. On Metrics and Measurements. In Eusgeld et al. [EFR08], pages 7–13. doi:10.1007/978-3-540-68947-8_2.

[BGK+96] Johan Bengtsson, David W. O. Griffioen, Kare J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Alur and Henzinger [AH96], pages 244–256. doi:10.1007/3-540-61474-5_73.

[BGTM98] Gunter Bolch, Stefan Greiner, Kishor Shridharbhai Trivedi, and Hermann de Meer. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation With Computer Science Applications*. John Wiley & Sons, Hoboken, NJ, USA, 1998. doi:10.1002/0471791571.

[BHS07] Frank Buschmann, Kevin Henney, and Douglas C. Schmidt. *Pattern-oriented software architecture, on patterns and pattern languages*, volume 5. John Wiley & Sons, Hoboken, NJ, USA, 2007.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. doi:10.1016/j.jss.2008.03.066.

[BM04] Simonetta Balsamo and Moreno Marzolla. UML-PSI: The UML Performance Simulator. In *Proceedings of the First International Conference on the Quantitative Evaluation of Systems (QEST)* [QES04], pages 340–341. doi:10.1109/QEST.2004.10008.

[Bon00] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd International Workshop on Software and Performance* [WOS00], pages 195–203. URL: http://doi.acm.org/10.1145/350391.350432, doi:10.1145/350391.350432.

[BR08] Rainer Böhme and Ralf Reussner. Validation of Predictions with Measurements. In Eusgeld et al. [EFR08], pages 14–18. doi:10.1007/978-3-540-68947-8_3.

[BSL16] Gunnar Brataas, E. Stav, and Sebastian Lehrig. Analysing Evolution of Work and Load. In *Proceedings of the 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)* [QoS16], pages 90–95. doi:10.1109/QoSA.2016.18.

[CdL12] Javier Cámara and Rogério de Lemos. Evaluation of Resilience in Self-adaptive Systems Using Probabilistic Model-checking. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* [SEA12], pages 53–62.

URL: http://dl.acm.org/citation.cfm?id=2666795.2666805, doi:10.1109/seams.2012.6224391.

[CdLG+09a] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Programming and Software Engineering*. Springer, Berlin/Heidelberg, Germany, 2009.

[CdLG+09b] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems* [CdLG+09a].

[CG12] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation: Self-Adaptive Systems. *Journal of Systems and Software*, 85(12):2860–2875, 2012. doi:10.1016/j.jss.2012.02.060.

[CGB11] Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors. *Proceedings of the 5th European Conference on Software Architecture (ECSA)*, volume 6903 of *Lecture Notes in Computer Science (LNCS)*, Berlin/Heidelberg, Germany, 2011. Springer.

[CGS09] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* [SEA09], pages 132–141. doi:10.1109/SEAMS.2009.5069082.

[CIL+07] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil Sahai. SLA Decomposition: Translating Service Level Objectives to System Level Thresholds. In *Proceedings of the 4th International Conference on Autonomic Computing (ICAC)* [ICA07], page 3. doi:10.1109/ICAC.2007.36.

[Cle96] Paul C. Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design* [IWS96], pages 16–25. URL: http://dl.acm.org/citation.cfm?id=857204.858261, doi:10.1109/iwssd.1996.501143.

[Clo14] *Proceedings of the 6th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2014.

[DB78] Peter J. Denning and Jeffrey P. Buzen. The Operational Analysis of Queueing Network Models. *ACM Computing Surveys*, 10(3):225–261, 1978. URL: http://doi.acm.org/10.1145/356733.356735, doi:10.1145/356733.356735.

[DD09] Armen Der Kiureghian and Ove Ditlevsen. Aleatory or epistemic? Does it matter? *Structural Safety*, 31(2):105–112, 2009. doi:10.1016/j.strusafe.2008.06.020.

[DEP12] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Model Transformations. In Bernardo et al. [BCP12], pages 91–136. URL: http://dx.doi.org/10.1007/978-3-642-30982-3_4, doi:10.1007/978-3-642-30982-3_4.

[dLGGG17] Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, editors. *Software Engineering for Self-Adaptive Systems (SEfSAS) 3*, volume 9640 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin/Heidelberg, Germany, 2017.

[DN02] Liliana Dobrica and Eila Niemela. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002. doi:10.1109/TSE.2002.1019479.

[EEKR00] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin/Heidelberg, Germany, 2000.

[EFR08] Irene Eusgeld, Felix C. Freiling, and Ralf Reussner, editors. *Dependability Metrics: Advanced Lectures*, volume 4909 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin/Heidelberg, Germany, 2008.

[FAOW+09] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced Modeling and Solution of Layered Queueing Net-

works. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009. doi:10.1109/TSE.2008.74.

[FAS+12] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the Cloud: What It Should, Can, and Cannot Be. In *Proceedings of the 4th Technology Conference on Performance Evaluation & Benchmarking* [TPC12], pages 173–188. doi:10.1007/978-3-642-36727-4_12.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Ehrig et al. [EEKR00], pages 296–309. URL: http://dx.doi.org/10.1007/978-3-540-46464-8_21, doi:10.1007/978-3-540-46464-8_21.

[FS09] Franck Fleurey and Arnor Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In Schürr and Selic [SS09], pages 606–621. URL: http://dx.doi.org/10.1007/978-3-642-04425-0_47, doi:10.1007/978-3-642-04425-0_47.

[GBB12] Thomas Goldschmidt, Steffen Becker, and Erik Burger. Towards a Tool-Oriented Taxonomy of View-Based Modelling. In Sinz and Schürr [SS12], pages 59–74. doi:10.1007/978-3-642-30412-5_1.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004. doi:10.1109/MC.2004.175.

[Gie07] Holger Giese. Modeling and Verification of Cooperative Self-adaptive Mechatronic Systems. In Kordon and Sztipanovits [KS07], pages 258–280. URL: http://dx.doi.org/10.1007/978-3-540-71156-8_14, doi:10.1007/978-3-540-71156-8_14.

[Gla98] Robert L. Glass. *Software Runaways: lessons learned from massive software project failures.* Prentice Hall, Upper Saddle River, NJ, USA, 1998.

[GMR09] Vincenzo Grassi, Raffaela Mirandola, and Enrico Randazzo. Model-Driven Assessment of QoS-Aware Self-Adaptation. In Cheng et al. [CdLG+09a], pages 201–222. URL: http://dx.doi.org/10.1007/978-3-642-02161-9_11, doi:10.1007/978-3-642-02161-9_11.

[Hal60] Paul Richard Halmos. *Naive Set Theory*. Springer, New York, NY, USA, 1960. doi:10.2307/2023253.

[HBK11] Nikolaus Huber, Fabian Brosig, and Samuel Kounev. Model-based self-adaptive resource allocation in virtualized environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* [SEA11], pages 90–99. doi:10.1145/1988008.1988021.

[Hei15] Christian Heinzemann. *Verification and Simulation of Self-Adaptive Mechatronic Systems*. Dissertation, Paderborn University, Paderborn, 2015.

[HGB10] Regina Hebig, Holger Giese, and Basil Becker. Making Control Loops Explicit when Architecting Self-adaptive Systems. In *Proceedings of the 2nd International Workshop on Self-Organizing Architectures (SOAR)* [SOA10], pages 21–28. URL: http://doi.acm.org/10.1145/1809036.1809042, doi:10.1145/1809036.1809042.

[HKR13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity: What it is, and What it is Not. In Kephart [Kep13].

[Hot14] *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability*, HotTopiCS, New York, NY, USA, 2014. ACM.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, UK, 2nd edition, 2004. doi:10.1017/cbo9780511810275.

[HWBK15] Nikolaus Huber, Jürgen Walter, Manuel Bähr, and Samuel Kounev. Model-based Autonomic and Performance-aware System Adaptation in Heterogeneous Resource Environments: A Case Study. In *Proceedings of*

the 2015 IEEE International Conference on Cloud and Autonomic Computing (ICCAC) [ICC15]. doi:10.1109/iccac.2015.27.

[ICA07] *Proceedings of the 4th International Conference on Autonomic Computing (ICAC)*, Washington, DC, USA, 2007. IEEE.

[ICC15] *Proceedings of the 2015 IEEE International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE, 2015.

[ICP12] *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, New York, NY, USA, 2012. ACM.

[ICP15] *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, New York, NY, USA, 2015. ACM.

[ICS04] *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Washington, DC, USA, 2004. IEEE.

[ICS06] *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, New York, NY, USA, 2006. ACM.

[ICS10] *3rd International Conference on Software Testing, Verification, and Validation – Workshops (ICSTW)*, Washington, DC, USA, 2010. IEEE.

[ILFL12] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a Consumer Can Measure Elasticity for Cloud Platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* [ICP12], pages 85–96. URL: http://doi.acm.org/10.1145/2188286.2188301, doi:10.1145/2188286.2188301.

[ITT15] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. A proactive approach for runtime self-adaptation based on queueing network fluid analysis. In *Proceedings of the 1st International Workshop on Quality-Aware DevOps* [QAD15], pages 19–24. doi:10.1145/2804371.2804375.

[IWS96] *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD)*, Washington, DC, USA, 1996. IEEE.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, Hoboken, NJ, USA, 1991.

[JW00] Prasad Jogalekar and Murray Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–603, 2000. doi:10.1109/71.862209.

[JYJ13] Daniel Jacobson, Danny Yuan, and Neeraj Joshi, 2013. URL: http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html.

[KC07] Barbara A. Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE-2007-01, Keele University and University of Durham, UK, 2007.

[KDJ04] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-Based Software Engineering. In *Proceedings of the 26th International Conference on Software Engineering* [ICS04], pages 273–281. URL: http://dl.acm.org/citation.cfm?id=998675.999432, doi:10.1109/icse.2004.1317449.

[Kep13] Jeffrey Kephart, editor. *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, Washington, DC, USA, 2013. IEEE.

[KHK+17] Jóakim Von Kistowski, Nikolas Herbst, Samuel Kounev, Henning Groenda, Christian Stier, and Sebastian Lehrig. Modeling and Extracting Load Intensity Profiles. *ACM Transactions on Autonomous and Adaptive Systems*, 11(4):23:1–23:28, 2017. URL: http://doi.acm.org/10.1145/3019596, doi:10.1145/3019596.

[Koz10] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010. URL: http://www.sciencedirect.com/science/article/pii/S016653160900100X, doi:10.1016/j.peva.2009.07.007.

[KS07] Fabrice Kordon and Janos Sztipanovits, editors. *Proceedings of the 12th Monterey Workshop on Reliable Systems on Unreliable Networked Platforms*, Lecture Notes in Computer Science (LNCS), Berlin/Heidelberg, Germany, 2007. Springer.

[KY95] George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice-Hall, Upper Saddle River, NJ, USA, 1995.

[LB15] Sebastian Lehrig and Steffen Becker. The CloudScale Method for Software Scalability, Elasticity, and Efficiency Engineering: A Tutorial. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* [ICP15], pages 329–331. URL: http://doi.acm.org/10.1145/2668930.2688818, doi:10.1145/2668930.2688818.

[LEB15] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. Scalability, Elasticity, and Efficiency in Cloud Computing: A Systematic Literature Review of Definitions and Metrics. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures* [QoS15], pages 83–92. URL: http://doi.acm.org/10.1145/2737182.2737185, doi:10.1145/2737182.2737185.

[Leh14] Sebastian Lehrig. Applying Architectural Templates for Design-Time Scalability and Elasticity Analyses of SaaS Applications. In *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability* [Hot14], pages 2:1–2:8. URL: http://doi.acm.org/10.1145/2649563.2649573, doi:10.1145/2649563.2649573.

[LGM13] Rogério Lemos, Holger Giese, and Hausi A. Müller, editors. *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin/Heidelberg, Germany, 2013.

[LMS03] Luciano Lavagno, Grant Martin, and Bran Selic, editors. *UML for Real: Design of Embedded Real-Time Systems*. Springer, Boston, MA, USA, 2003.

[Luc13] Markus Luckey. *Adaptivity Engineering: Modeling and Quality Assurance for Self-Adaptive Systems*. Dissertation, University of Paderborn, Paderborn, 2013.

[Mar02] John J. Marciniak, editor. *Encyclopedia of Software Engineering*. John Wiley & Sons, New York, NY, USA, 2002. doi:10.1002/0471028959.

[Mei58] Torben Meisling. Discrete-Time Queuing Theory. *Operations Research*, 6(1):96–105, 1958. URL: http://dx.doi.org/10.1287/opre.6.1.96, doi:10.1287/opre.6.1.96.

[Mey11] Joachim Meyer. Modellgetriebene Skalierbarkeitsanalyse von selbstadaptiven komponentenbasierten Softwaresystemen in der Cloud. Master's thesis, University of Paderborn, Paderborn, 2011.

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology, 2011. URL: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, doi:10.6028/nist.sp.800-145.

[MLL04] Seong-ick Moon, K. H. Lee, and Doheon Lee. Fuzzy branching temporal logic: Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 34(2):1045–1055, 2004. doi:10.1109/TSMCB.2003.819485.

[Mor99] Ana Moreira, editor. *Object-Oriented Technology. ECOOP'99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin/Heidelberg, Germany, 1999. doi:10.1007/3-540-46589-8.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. doi:10.1109/32.825767.

[Mur04] Richard Murch. *Autonomic Computing*. IBM Press, 2004.

[MV00] Daniel A. Menasce and A. F. Almeida Virgilio. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning.* Prentice Hall, Upper Saddle River, NJ, USA, 2000.

[ÖGW+14] P. O. Östberg, H. Groenda, S. Wesner, J. Byrne, D. S. Nikolopoulos, C. Sheridan, J. Krzywda, A. Ali-Eldin, J. Tordsson, E. Elmroth, C. Stier, K. Krogmann, J. Domaschka, C. B. Hauser, P. J. Byrne, S. Svorobej, B. Mccollum, Z. Papazachos, D. Whigham, S. Rüth, and D. Paurevic. The CACTOS Vision of Context-Aware Cloud Topology Optimization and Simulation. In *Proceedings of the 6th International Conference on Cloud Computing Technology and Science (CloudCom)* [Clo14], pages 26–31. doi:10.1109/CloudCom.2014.62.

[PC13] J. F. Pérez and G. Casale. Assessing SLA Compliance from Palladio Component Models. In *Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* [SYN13], pages 409–416. doi:10.1109/SYNASC.2013.60.

[Pla16] Marie Christin Platenius. *Fuzzy Matching of Comprehensive Service Specifications.* Dissertation, Paderborn University, 2016.

[PPMMG10] Diego Perez-Palacin, Raffaela Mirandola, José Merseguer, and Vincenzo Grassi. QoS-Based Model Driven Assessment of Adaptive Reactive Systems. In *3rd International Conference on Software Testing, Verification, and Validation – Workshops (ICSTW)* [ICS10], pages 299–308. doi:10.1109/ICSTW.2010.20.

[Pre99] Lutz Prechelt. *Rolle und Methodik kontrollierter Experimente in der Softwaretechnik.* Habilitationsschrift, Universität Karlsruhe, Karlsruhe, 1999.

[PRH10] Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors. *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 6394 of *Lecture Notes in Computer Science (LNCS)*, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-16145-2.

[QAD15] *Proceedings of the 1st International Workshop on Quality-Aware DevOps*, New York, NY, USA, 2015. ACM.

[QES04] *Proceedings of the First International Conference on the Quantitative Evaluation of Systems (QEST)*, Washington, DC, USA, 2004. IEEE.

[QoS15] *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, New York, NY, USA, 2015. ACM.

[QoS16] *Proceedings of the 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, New York, NY, USA, 2016. ACM.

[QRA16] *Proceedings of the 1st Workshop on Qualitative Reasoning about Software Architectures (QRASA)*, Washington, DC, USA, 2016. IEEE.

[RBB+11] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical Report 2190-4782, Karlsruhe Institute of Technology, 2011. doi:10.5445/IR/1000022503.

[RF08] Ralf Reussner and Viktoria Firus. Basic and Dependent Metrics. In Eusgeld et al. [EFR08], pages 37–38. URL: http://dx.doi.org/10.1007/978-3-540-68947-8_5, doi:10.1007/978-3-540-68947-8_5.

[Sal07] Simona Salicone. *Measurement Uncertainty: An Approach via the Mathematical Theory of Evidence*. Springer Series in Reliability Engineering. Springer, Boston, MA, USA, 2007.

[SAV+16] Bradley Schmerl, Jesper Andersson, Thomas Vogel, Myra Cohen, Cecilia M. F. Rubira, Yuriy Brun, Alessandra Gorla, Franco Zambonelli, and Luciano Baresi. Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems. In de Lemos et al. [dLGGG17].

[SBW99] Clemens Szyperski, Jan Bosch, and Wolfgang Weck. Component-Oriented Programming. In Moreira [Mor99], pages 184–192. URL: http://dx.

doi.org/10.1007/3-540-46589-8_10, doi:10.1007/3-540-46589-8_10.

[SE.11] *Proceedings of the Software Engineering (SE)*, 2011.

[SEA09] *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2009.

[SEA11] *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, New York, NY, USA, 2011. ACM.

[SEA12] *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Piscataway, NJ, USA, 2012. IEEE.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Upper Saddle River, NJ, USA, 1996.

[SHK98] Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the Cost of Software Quality. *Communications of the ACM*, 41(8):67–73, 1998. URL: http://doi.acm.org/10.1145/280324.280335, doi:10.1145/280324.280335.

[SK16] Christian Stier and Anne Koziolek. Considering Transient Effects of Self-Adaptations in Model-Driven Performance Analyses. In *Proceedings of the 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)* [QoS16], pages 80–89. doi:10.1109/QoSA.2016.14.

[Smi90] Connie U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, Boston, MA, USA, 1990.

[SOA10] *Proceedings of the 2nd International Workshop on Self-Organizing Architectures (SOAR)*, New York, NY, USA, 2010. ACM.

[SS09] Andy Schürr and Bran Selic, editors. *Model Driven Engineering Languages and Systems*, volume 5795 of *Programming and Software Engineering*. Springer, Berlin/Heidelberg, Germany, 2009.

[SS12] Elmar J. Sinz and Andy Schürr, editors. *Modellierung*, volume 201 of *Lecture Notes in Informatics (LNI)*, Bonn, 2012. Bonner Köllen Verlag. doi:10.1007/978-3-642-30412-5_1.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, New York, NY, USA, 2006.

[SW03] Connie U. Smith and Lloyd G. Williams. Software Performance Engineering. In Lavagno et al. [LMS03], pages 343–365. URL: http://dx.doi.org/10.1007/0-306-48738-1_16, doi:10.1007/0-306-48738-1_16.

[SYN13] *Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2013.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, MA, USA, 2nd edition, 2002.

[TBv04] Jos J.M. Trienekens, Jacques J. Bouman, and Mark van der Zwan. Specification of Service Level Agreements: Problems, Principles and Practices. *Software Quality Journal*, 12(1):43–57, 2004. URL: http://dx.doi.org/10.1023/B:SQJO.0000013358.61395.96, doi:10.1023/B:SQJO.0000013358.61395.96.

[TN07] Vicenç Torra and Yasuo Narukawa. *Modeling Decisions: Information Fusion and Aggregation Operators*. Springer, Berlin/Heidelberg, Germany, 2007.

[TPC12] *Proceedings of the 4th Technology Conference on Performance Evaluation & Benchmarking (TPCTC)*, 2012.

[vBCR02] Rini van Solingen, Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric (GQM) Approach. In Marciniak [Mar02]. doi:10.1002/0471028959.sof142.

[vDL98] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998. doi:10.1109/32.730542.

[VG12] Thomas Vogel and Holger Giese. A Language for Feedback Loops in Self-adaptive Systems: Executable Runtime Megamodels. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* [SEA12], pages 129–138. URL: http://dl.acm.org/citation.cfm?id=2666795.2666816, doi:10.1109/seams.2012.6224399.

[VG14] Thomas Vogel and Holger Giese. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems*, 8(4):18:1, 2014. doi:10.1145/2555612.

[vMvH11] Robert von Massow, André van Hoorn, and Wilhelm Hasselbring. Performance Simulation of Runtime Reconfigurable Component-Based Software Architectures. In Crnkovic et al. [CGB11], pages 43–58. URL: http://dx.doi.org/10.1007/978-3-642-23798-0_5, doi:10.1007/978-3-642-23798-0_5.

[vWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* [ICP12], pages 247–248. URL: http://doi.acm.org/10.1145/2188286.2188326, doi:10.1145/2188286.2188326.

[WOS00] *Proceedings of the 2nd International Workshop on Software and Performance (WOSP)*, New York, NY, USA, 2000. ACM.

[WSB+10] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. RELAX: a language to address uncertainty in self-adaptive systems requirement: Requirements Engineering. *Requirements Engineering*, 15(2):177–196, 2010. doi:10.1007/s00766-010-0101-0.

[WSG+13] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. On patterns for decentralized control in

self-adaptive systems. In Lemos et al. [LGM13], pages 76–107. doi:10.1007/978-3-642-35813-5_4.

[Zad65] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965. doi:10.1142/9789814261302_0021.

[Zad08] Lotfi A. Zadeh. Is There a Need for Fuzzy Logic? *Information Sciences*, 178(13):2751–2779, 2008. URL: http://dx.doi.org/10.1016/j.ins.2008.02.012,TitelanhanddieserDOIinCitavi-Projektübernehmen, doi:10.1109/nafips.2008.4531354.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based Development of Dynamically Adaptive Software. In *Proceedings of the 28th International Conference on Software Engineering* [ICS06], pages 371–380. URL: http://doi.acm.org/10.1145/1134285.1134337, doi:10.1145/1134285.1134337.

Websites and Standards

[CS16] Shang-Wen Cheng and Bradley Schmerl. Model Problem: Znn.com, 2016. URL: https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com/ [cited 26.09.2016].

[Fac16] Inc. Facebook. Company Info, 2016. URL: http://newsroom.fb.com/company-info/ [cited 16.11.2016].

[Goo16] Google. Google Scholar, 2016. URL: https://scholar.google.de/.

[Hai13] Cameron Haight. Enter Web-scale IT, 2013. URL: http://blogs.gartner.com/cameron_haight/2013/05/16/enter-web-scale-it/ [cited 23.12.2016].

[Kar16a] Karlsruhe Institute of Technology. Palladio Tools, 2016. URL: http://www.palladio-simulator.com/tools/ [cited 16.11.2016].

[Kar16b] Karlsruhe Institute of Technology. Software Architecture Analysis, 2016. URL: http://www.palladio-simulator.com/analysis/ [cited 09.10.2016].

[Net16] Inc. Netflix. Netflix Company Profile, 2016. URL: https://ir.netflix.com/ [cited 29.09.2016].

[Obj16] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation, 2016. URL: http://www.omg.org/spec/QVT/ [cited 16.11.2016].

[Sto16] Luke Stone. Bringing Pokémon GO to life on Google Cloud, 2016. URL: https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html [cited 02.10.2016].

[The16a] The Eclipse Foundation. Eclipse - The Eclipse Foundation open source community website, 2016. URL: https://www.eclipse.org/ [cited 16.11.2016].

List of Figures

1.1. Znn.com components 7
1.2. Alternative system architecture configurations. 9
1.3. Scientific contribution 13

2.1. Example membership function for a fuzzy set. 19
2.2. Probability density function $P(x)$ for a standard normal distribution. . . 20
2.3. Imprecise definition of "fast", modeled with a fuzzy set. 22
2.4. MAPE-K feedback loop [Mur04]. 27

3.1. Model that defines the taxonomy of view-based modeling [GBB12]. . . . 33
3.2. Basic Concept of model transformation [DEP12]. 34
3.3. Model-Driven Software Performance Engineering process. 37
3.4. Comparision of MDSPE analysis types. 38
3.5. Overview of PCM partial models and model transformations. 40

4.1. Venn diagram of related work. 55
4.2. Feature diagram for classification of related work. 56
4.3. Viewpoints in SimuLizar's self-adaptive sys. perf. modeling approach . . 62
4.4. Package diagram of SimuLizar's modeling packages. 65
4.5. Use case diagrams for comparison of modeling roles. 66
4.6. UML activity diagram of the modeling process in SimuLizar. 68
4.7. System architecture types for the Znn.com system. 71
4.8. PCM repository meta model excerpt. [RBB+11] 73
4.9. System resource context for the Znn.com system. 74
4.10. PCM resource environment meta model excerpt. [RBB+11] 75
4.11. Initial system architecture configuration for the Znn.com system. 78
4.12. PCM System meta model. [RBB+11] 79
4.13. Initial deployment for the Znn.com system. 80
4.14. PCM Allocation meta model. [RBB+11] 81
4.15. Usage context with one usage scenario for the Znn.com system. 82
4.16. Time-dependent variation of the load in the Znn.com example. 83

4.17. PCM Usage Model meta model [RBB+11] 84
4.18. Combination of Usage Evolution [BSL16] and DLIM meta model [KHK+17] 85
4.19. Service level objectives for our Znn.com system. 88
4.20. Service level objective meta model. 88
4.21. Monitor repository for the Znn.com system. 90
4.22. Monitor repository meta model 91
4.23. Reconfiguration rule model for the Znn.com system. 93
4.24. Action element of the “scale out” tactic for the Znn.com system. 95
4.25. Action element of the “textMode” tactic for the Znn.com system. 97
4.26. Reconfiguration rule meta model 98
4.27. Evaluation process. 103

5.1. Venn diagram of related work. 117
5.2. Feature diagram for classification of related work. 118
5.3. Overview of the prediction methods 122
5.4. Example trace of self-adaptive system states. 129
5.5. Accuracy and time range dimensions of requirement relaxation. 132
5.6. Excerpts of a self-adaptive system state space. 133
5.7. Membership function with for SLOs with exact accuracy. 138
5.8. Membership function with for SLOs with strict accuracy. 139
5.9. Membership function with for SLOs with tolerant accuracy. 141
5.10. Membership function with for SLOs with vague accuracy. 143
5.11. Exploration of scalability 150
5.12. Comparison of the predictions and measurements in a time series. . . . 163
5.13. State trace of the Znn.com system. 164
5.14. Comparison of the predictions and measurements in a box plot. 165

6.1. Component diagram showing the SimuLizar architecture. 173
6.2. Overview of packages and dependencies. 174
6.3. Screenshot of SimuLizar Bench. 176
6.4. System Type Viewpoint and its partial models in SimuLizar. 178
6.5. Run-Time Viewpoint and its partial models in SimuLizar. 179
6.6. Self-Adaptation Viewpoint and its partial models in SimuLizar. 180
6.7. Comparison between generator-based and interpreter-based MDSPE. . . 181

6.8. Self-Adaptive System Performance Analysis in SimuLizar Bench. 182
6.9. Interpretation of a performance model. 183
6.10. Monitor repository for the Znn.com system. 184
6.11. MAPE-K feedback loop in SimuLizar. 186
6.12. SLO thresholds visualization in SimuLizar. 189
6.13. SLO achievement visualization in SimuLizar. 190

I.1. Znn.com system type elements (1/2) 228
I.2. Znn.com system type elements (2/2) 228
I.3. Znn.com system resource context 229
I.4. Znn.com initial system architecture configuration 229
I.5. Znn.com initial system deployment 229
I.6. Znn.com system usage context 230
I.7. Znn.com service lvel objective SLO_{MRT} 230
I.8. Znn.com monitor repository 231
I.9. Znn.com reconfigurations 231
I.10. Znn.com reconfiguration action addApplicationServer 233
I.11. Znn.com reconfiguration action removeApplicationServer 233

List of Tables

4.1. Feature Configurations of Related Modeling Approaches 58
4.2. Elements in the Modeling Viewpoints 64
4.3. Evaluation Goal 101
4.4. Question 1: Applicability 101
4.5. Question 2: Limitations 102

5.1. Evaluation Results of Related Prediction Methods 119
5.2. Evaluation Goal 160
5.3. Question 1: Applicability 161
5.4. Comparison of the Scalability and Elasticity Metrics 163
5.5. Values of the Znn.com state trace 164

List of Listings

4.1. Precondition “mrtGreater2s” for “scaleOut” Reconfiguration Tactic . . . 94

5.1. Scalability Prediction Method 149
5.2. Elasticity Prediction Method 156

I.1. Znn.com Reconfiguration Precondition mrtGreater2s 231
I.2. Znn.com Reconfiguration Precondition mrtLower2s 232

Complete Znn.com Example

I.1. System Element Type View

I.1.1. System Architecture Types

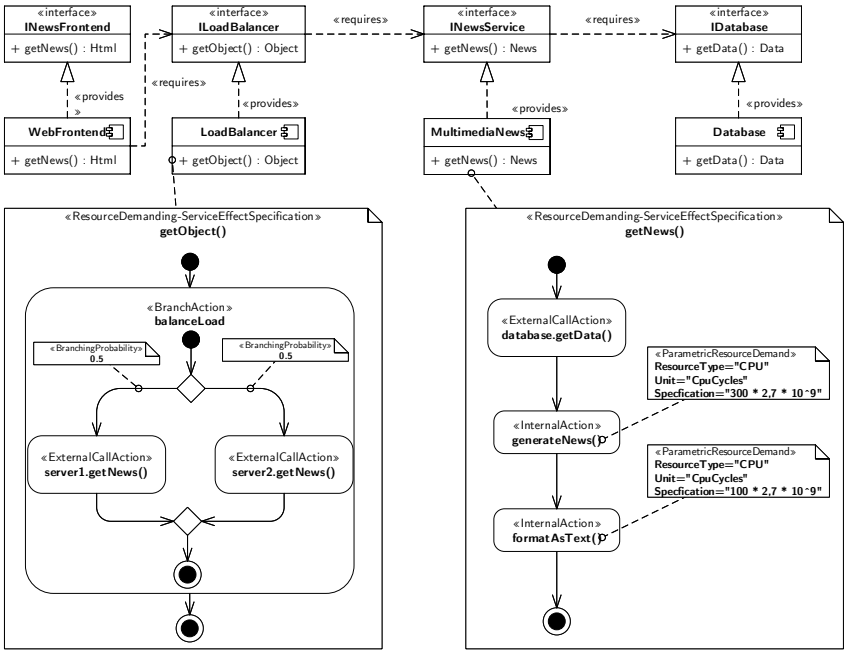


Figure I.1.: Znn.com system type elements (1/2)

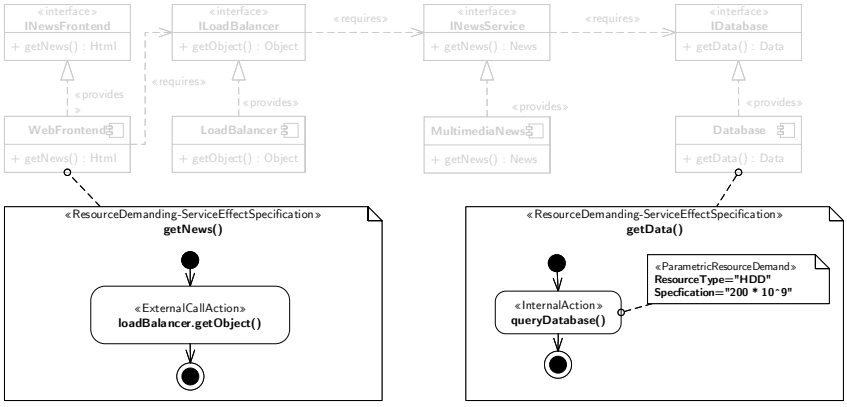


Figure I.2.: Znn.com system type elements (2/2)

I.1.2. System Resource Context

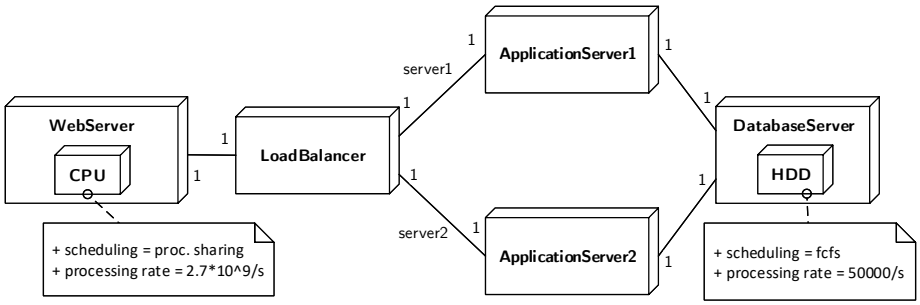


Figure I.3.: Znn.com system resource context

I.2. Initial Configuration View

I.2.1. Initial System Architecture Configuration

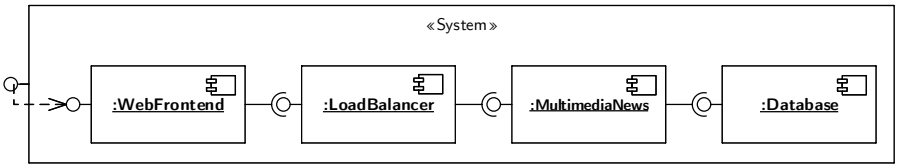


Figure I.4.: Znn.com initial system architecture configuration

I.2.2. Initial System Deployment

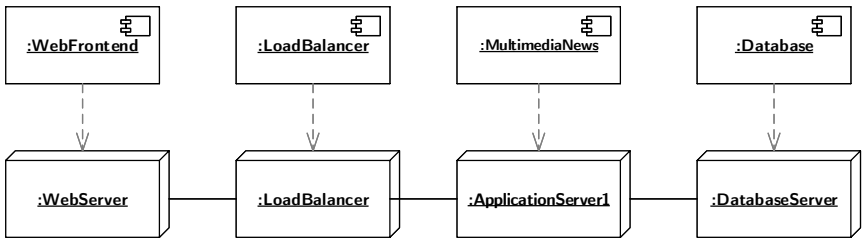


Figure I.5.: Znn.com initial system deployment

I.2.3. System Usage Context

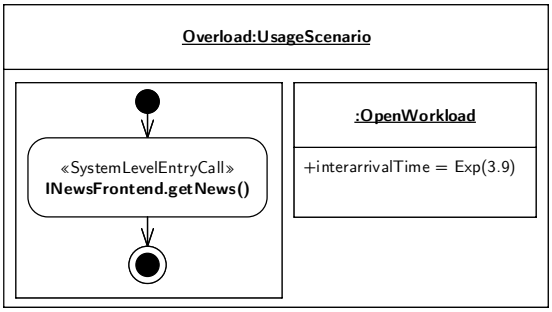


Figure I.6.: Znn.com system usage context

I.3. Reconfiguration View

I.3.1. Service Level Objectives

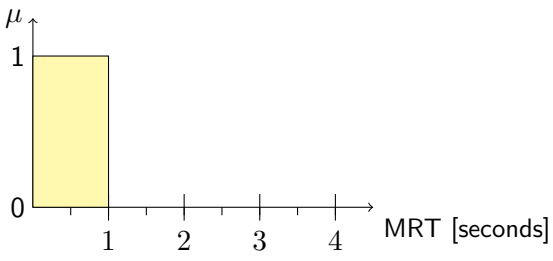


Figure I.7.: Znn.com service lvel objective SLO_{MRT}

I.3.2. Monitor Repository

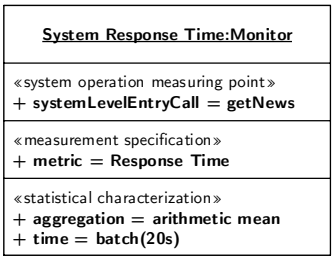


Figure I.8.: Znn.com monitor repository

I.3.3. Reconfigurations

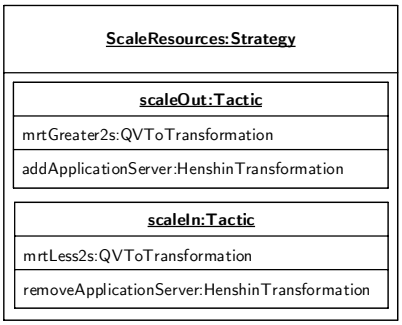


Figure I.9.: Znn.com reconfigurations

Listing I.1: Znn.com Reconfiguration Precondition mrtGreater2s

```
1 property threshold : Real = 1.0;
2 main() {
3   assert fatal(run-timeMeasurement.rootObjects()[Run-timeMeasurement
        ]->size() > 0)
4     with log ("No Measurements found!");
5
6   assert error (run-timeMeasurement.rootObjects()[Run-timeMeasurement
        ]->checkCondition() = true)
7     with log ("No reconfiguration required");
8 }
9 helper Set(Run-timeMeasurement) :: checkCondition() : Boolean {
10  self->forEach(measurement) {
```

```
11   log('Measured value is ' + measurement.measuringValue.toString());
12     if (measurement.measuringValue > threshold) {
13       log('Threshold is exceeded');
14       return true;
15     };
16   };
17   return false;
18 }
```

Listing I.2: Znn.com Reconfiguration Precondition mrtLower2s

```
1 property threshold : Real = 0.0;
2 main() {
3   assert fatal(run-timeMeasurement.rootObjects()[Run-timeMeasurement
       ]->size() > 0)
4     with log ("No Measurements found!");
5
6   assert error (run-timeMeasurement.rootObjects()[Run-timeMeasurement
       ]->checkCondition() = true)
7     with log ("No reconfiguration required");
8 }
9 helper Set(Run-timeMeasurement) :: checkCondition() : Boolean {
10   self->forEach(measurement) {
11   log('Measured value is ' + measurement.measuringValue.toString());
12     if (measurement.measuringValue < threshold) {
13       log('Threshold is exceeded');
14       return true;
15     };
16   };
17   return false;
18 }
```

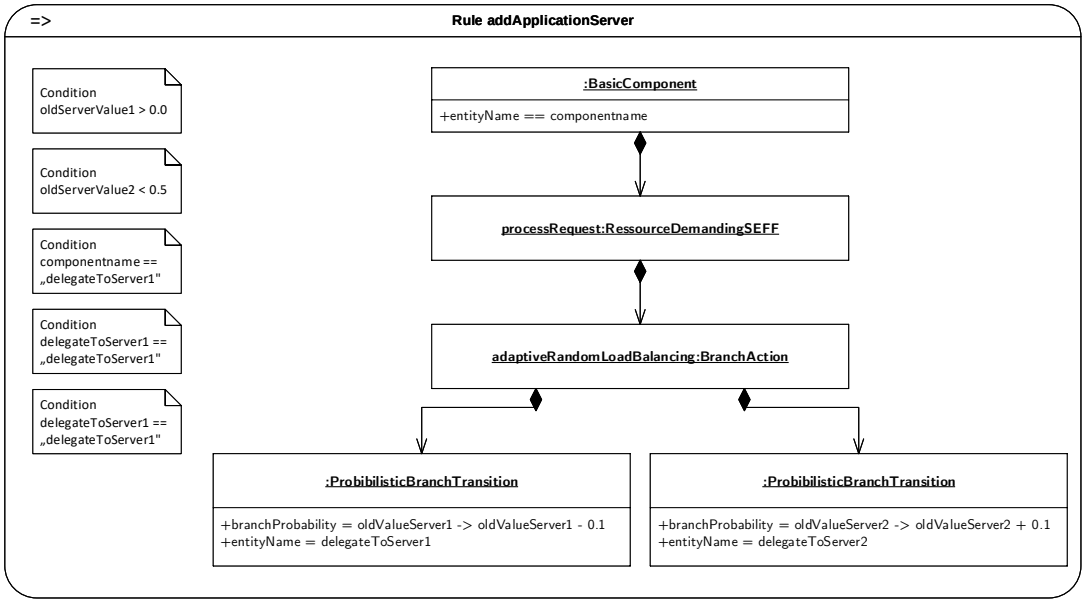


Figure I.10.: Znn.com reconfiguration action addApplicationServer

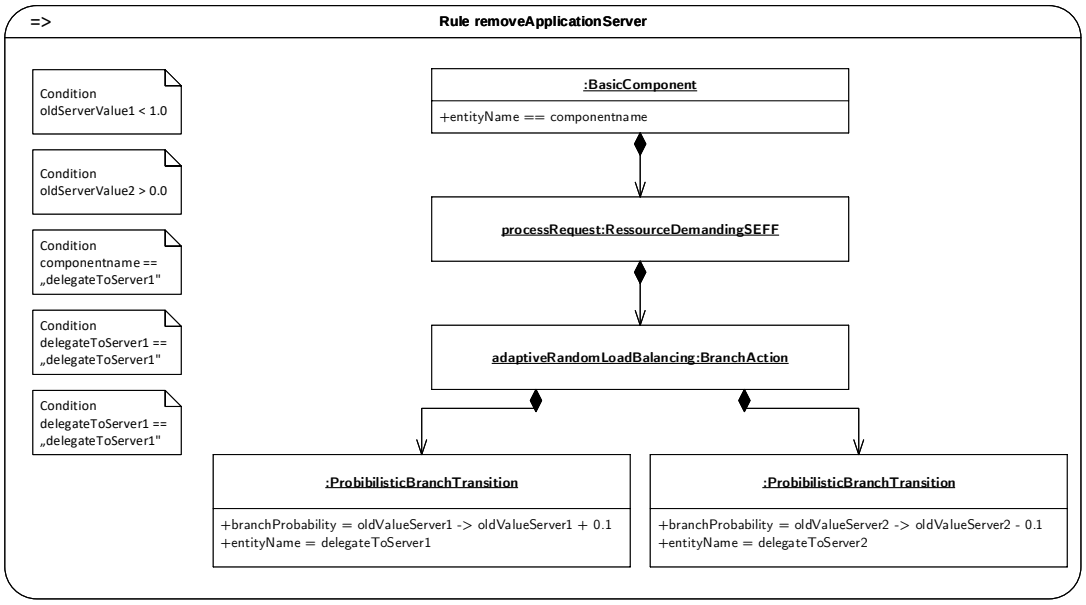


Figure I.11.: Znn.com reconfiguration action removeApplicationServer

II

Auxiliary Formalization

Definition II.1 (Monitoring Labels) *A self-adaptive system state transition* $(a_x, m_x, s_x, t_x) \rightarrow (a_y, m_y, s_y, t_t)$ *is labeled with label* l_Δ *if and only if* $m_x \neq m_y$. *The transition can then be written as* $\Sigma_x \xrightarrow{\Delta} \Sigma_y$.

Definition II.2 (Reconfiguration Labels) *A self-adaptive system state transition* $(a_x, m_x, s_x, t_x) \rightarrow (a_y, m_y, s_y, t_t)$ *is labeled with label* l_Δ *if and only if* $a_x \neq a_y$. *The transition can then be written as* $\Sigma_x \xrightarrow{\alpha} \Sigma_y$.

Definition II.3 (Context Change Labels) *A self-adaptive system state transition* $(a_x, m_x, s_x, t_x) \rightarrow (a_y, m_y, s_y, t_t)$ *is labeled with label* l_Δ *if and only if* $s_x \neq s_y$. *The transition can then be written as* $\Sigma_x \xrightarrow{\sigma} \Sigma_y$.

Definition II.4 (Real Number Expression) *Let* V *be a set of real number variables. We define* $Exp(V)$ *the set of* real number expressions *over* V. *Each* $exp \in Exp(V)$ *is recursively defined by the rules:*

$$exp := x|v|(exp)|exp \sim exp$$

for $x \in \mathbb{R}$, $v \in V$, *and* $\sim \in \{+, -, *, /\}$. *(cf. [HR04, p. 260])*

Definition II.5 (Real Number Variable Constraint)
Let V be a set of real number variables. A real number variable constraint *ψ is a conjunctive formula of atomic real number variable constraints of the form $v \sim exp$ for $v \in V$, $\sim \in \{<, \leq, =, \geq, >\}$ and $exp \in Exp(V)$. We use $\Psi(V)$ to denote the set of real number variable constraints. (cf. [BGK+96])*

Definition II.6 (Real Number Variable Constraint Vector)
Let $\Psi(V)$ be a set of real number variable constraints. A real number variable constraint vector *π is a vector of atomic real number variable constraints of the form $\pi = \langle \psi_0, \psi_1, \psi_2, \ldots, \psi_n \rangle$ for $\psi_i \in \Psi(V)$. We use $\Pi(V)$ to denote the set of real number variable constraint vectors.*
We define the Boolean truth value of a real number variable constraint vector $\pi = \langle \psi_0, \psi_1, \psi_2, \ldots, \psi_n \rangle$ as the Boolean truth value of $\psi_0 \vee \psi_1 \vee \psi_2 \vee \ldots \vee \psi_n$.