Model-Driven Online Capacity Management for Component-Based Software Systems

Dissertation

Dipl.-Inform. André van Hoorn

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2014

Kiel Computer Science Series (KCSS) 2014/6 v1.0 dated 2014-10-10

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via http://www.informatik.uni-kiel.de/kcss

Published by the Department of Computer Science, Kiel University

Software Engineering Group

Please cite as:

▻ André van Hoorn. *Model-Driven Online Capacity Management for Component-Based Software Systems.* Number 2014/6 in Kiel Computer Science Series. Department of Computer Science, 2014. Dissertation, Faculty of Engineering, Kiel University.

```
@book{vanHoorn2014Dissertation,
  author    = {Andr\'e van Hoorn},
  title     = {Model-Driven Online Capacity Management
               for Component-Based Software Systems},
  publisher = {Department of Computer Science, Kiel University},
  address   = {Kiel, Germany}
  year      = {2014},
  number    = {2014/6},
  isbn      = {978-3-7357-5118-8},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering, Kiel University}
}
```


Herstellung und Verlag: BoD — Books on Demand, Norderstedt

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Examiner:	Prof. Dr. Wilhelm Hasselbring Kiel University
2. Examiner:	Prof. Dr. Ralf Reussner Karlsruhe Institute of Technology
3. Examiner:	Prof. Dr. Samuel Kounev University of Würzburg

Date of the oral examination (disputation): September 18, 2014

Abstract

Capacity management is a core activity when designing and operating distributed software systems. It comprises the provisioning of data center resources and the deployment of software components to these resources. The goal is to continuously provide adequate capacity, i. e., service level agreements should be satisfied while keeping investment and operating costs reasonably low. Traditional capacity management strategies are rather static and pessimistic: resources are provisioned for anticipated peak workload levels. Particularly, enterprise application systems are exposed to highly varying workloads, leading to unnecessarily high total cost of ownership due to poor resource usage efficiency caused by the aforementioned static capacity management approach.

During the past years, technologies emerged that enable dynamic data center infrastructures—e. g., leveraged by cloud computing products. These technologies build the foundation for elastic online capacity management, i. e., adapting the provided capacity to workload demands based on a short-term horizon. Because manual online capacity management is not an option, automatic control approaches have been proposed. However, most of these approaches focus on coarse-grained adaptation actions and adaptation decisions are based on aggregated system-level measures. Architectural information about the controlled software system is rarely considered.

This thesis introduces a model-driven online capacity management approach for distributed component-based software systems, called SLAstic. The core contributions of this approach are *a*) modeling languages to capture relevant architectural information about a controlled software system, *b*) an architecture-based online capacity management framework based on the common MAPE-K control loop architecture, *c*) model-driven techniques supporting the automation of the approach, *d*) architectural runtime reconfiguration operations for controlling a system's capacity, *e*) as well as an integration of the Palladio Component Model. A qualitative and quantitative evaluation of the approach is performed by case studies, lab experiments, and simulation.

Preface

by Prof. Dr. Wilhelm Hasselbring

Typical web-based information systems face workloads that highly vary over time. The full capacity for meeting SLAs (Service Level Agreements) at peek workloads is not required during times with low workload. To reduce operation costs, resources may be allocated dynamically at runtime. Such elastic cloud-based systems face the great challenge of meeting SLAs while allocating resources dynamically. Appropriate monitoring and forecasting/prediction techniques are required.

In this thesis, André van Hoorn designs, implements and evaluates the innovative, architecture-based approach SLAstic to meet defined SLAs in elastic, cloud-based systems. Besides the conceptual work, this work contains a significant experimental part with an implementation and a multifaceted evaluation. This engineering dissertation has been extensively evaluated with simulations and lab experiments, including data from industrial systems.

Specific contributions are the SLAstic meta-model for architecture modeling and the SLAstic control component consisting of a model updater, a performance evaluator, a workload forecaster, a performance predictor and an adaptation planner. The design and implementation includes the Kieker monitoring framework as a major contribution.

From a software engineering perspective, it is remarkable that model-driven software engineering techniques are realized for model-driven instrumentation of the system under control, for updating the observed system model, and for generating simulations, including a transformation to the Palladio Component Model.

If you are interested in operating cloud-based systems, this is a recommended reading for you.

Wilhelm Hasselbring
Kiel, September 2014

Contents

Abstract v

Preface vii

1 Introduction 1

I Foundations 9

2 Model-Driven Software Engineering 11

3 Software System Architecture: Description and Reconfiguration 21

4 Quality of Service Evaluation and Capacity Management 39

II SLAstic Approach 73

5 Research Design 75

6 Architectural Modeling 91

7 Kieker Framework 107

8 SLAstic Framework 125

9 Model-Driven Online Capacity Management 143

10 Runtime Reconfiguration for Controlling Capacity 161

11 Utilizing the Palladio Component Model in SLAstic 173

III Evaluation **199**

12 Industrial Case Study **201**

13 Lab Experiments **225**

14 Simulation-Based Evaluation **247**

15 Reviewing Kieker's History, Development, and Impact **259**

16 Related Work **279**

IV Conclusions & Future Work **293**

17 Conclusions **295**

18 Future Work **299**

List of Acronyms **305**

List of Figures **311**

List of Tables **315**

Bibliography **317**

Chapter 1

Introduction

This chapter provides an introduction to this thesis by stating the motivation for this research (Section 1.1), summarizing the approach and its contributions (Section 1.2), and outlining the document structure (Section 1.3).

1.1 Motivation and Problem Statement

Quality of service (QoS) measures quantify the degree to which a software system's runtime quality properties—with respect to characteristics like performance, reliability, and availability—satisfy respective requirements or user expectations [Becker et al., 2006b; ISO/IEC, 2005a]. Poor QoS causes considerable costs, e. g., by losing customers that switch to alternative system, or by penalties being due for payment to third parties. With respect to the latter, such penalties are typically specified as part of so-called service level agreements (SLAs) [IBM, 2003; Tosic, 2004]—particularly among partners providing and consuming business-critical services. For example, an SLA may define the penalty that is due if a specified software service violates its QoS measures in terms of maximum response times or service availability over a certain period of time. Hence, capacity management—also referred to as capacity planning—is an important activity when designing and operating a software system [ISO/IEC, 2005a]. The goal is that a system provides adequate capacity, i. e., that it continuously satisfies its SLAs.

Traditional capacity planning strategies follow a rather static approach [Jain, 1991; Menascé and Almeida, 2002]: data center resources are provisioned for peak workload levels, based on long-term workload characterization and forecasting. However, particularly in enterprise application systems (EASs), workload is highly varying [Arlitt et al., 2001; Gmach et al., 2007; Goševa-Popstojanova et al., 2006]. This includes time-dependent

variations in workload intensity, e. g., based on the time of day and day of week, as well as changes in request patterns contained in a system's operational profile [Musa, 1993]. As a consequence of static capacity planning and varying workloads, data center resources are poorly utilized [Barroso and Hölzle, 2007]. This underutilization of resources leads to an unnecessarily high total cost of ownership, particularly caused by increasing energy costs—e. g., needed for running the technical data center infrastructure comprising servers, network components, cooling, etc. As a measure of a system's resource usage economy, resource efficiency is becoming an increasingly important quality characteristic of software systems.

During the past years, virtualization technologies emerged and became commodity—e. g., in form of cloud computing services [Mell and Grance, 2011; Armbrust et al., 2009]. Infrastructure as a service (IaaS) products enable dynamic changes to virtual data center infrastructures by allocating and releasing resources—including virtual machines, storage, and database management systems—at runtime on a pay-per-use basis. Prominent examples are the commercial products Amazon Web Services [Amazon Web Services, Inc., 2014] and Windows Azure [Microsoft, Inc., 2014], as well as the open source products Eucalyptus [Nurmi et al., 2009; Eucalyptus Systems, Inc., 2014] and OpenStack [OpenStack Foundation, 2014]. Dynamic infrastructures build the technical foundation for online capacity management aiming for increased resource efficiency by adapting the provisioned infrastructure to current demands in a short-term manner. Of course, manual approaches to online capacity management are not an option. Automatic approaches for online capacity management have been proposed. However, most of these approaches are limited to coarse-grained adaptation actions, e. g., live-migration of virtual machine images. Architectural information about the controlled software is rarely considered. Moreover, decision making is based on aggregated system-level performance metrics, such as CPU utilization, considering application-level QoS measures, such as response times, only implicitly. In most cases, explicit architectural information about the controlled software system, including software components and their interactions, is missing. On the other hand, approaches for predicting a software system's QoS based on architectural models have been proposed, e. g., focusing on performance of component-based software systems [Koziolek, 2010; Bertolino and Mirandola, 2004; Becker et al., 2009]. However, these approaches mainly focus on design-time prediction.

1.2 Overview of Approach and Contributions

This thesis describes a model-driven online capacity management approach for distributed component-based software systems, called SLAstic. Relevant information about the controlled software system—including QoS-relevant structural and behavior aspects, as well as QoS requirements (SLAs)—are captured in architectural models. These models are used to initialize and execute an online capacity management infrastructure for continuous monitoring, analysis, and adaptation of the controlled system. The system is adapted by architectural runtime reconfiguration on system level and application level. This work draws from the areas of model-driven software engineering (MDSE) [Stahl and Völter, 2006; Brambilla et al., 2012], software performance engineering [Woodside et al., 2007; Cortellessa et al., 2011], software architecture [Taylor et al., 2009], and self-adaptive software systems [Salehie and Tahvildari, 2009].

In addition to the overall approach, the core contributions of this thesis can be grouped into six categories, according to the following Sections 1.2.1 to 1.2.6. Note that a more detailed summary of results—including those results achieved in the context of this thesis but not detailed here—is also provided in Section 5.2 as part of the description of the research design. Supplementary material for this thesis is publicly available online [van Hoorn, 2014].

1.2.1 Architectural Modeling

We selected and developed a set of complementary modeling languages supporting the SLAstic approach. The SLAstic meta-model allows to represent structural and behavioral aspects on a component-based software system's architecture, using an abstraction level close to the Palladio Component Model (PCM) [Becker et al., 2009]. As a result of joint work, we developed the meta-model agnostic modeling approaches MAMBA and S/T/A. Building on the OMG's SMM meta-model [Object Management Group, Inc., 2012b], MAMBA serves to attach performance measures to SLAstic models. S/T/A is employed to express reconfiguration plans. Instances of these modeling languages serve as a basis for monitoring instrumentation, framework initialization, and for the online analysis at runtime. For each of the modeling languages, an implementation is provided.

1.2.2 Online Capacity Management Framework

We developed two complementary frameworks, called Kieker and SLAstic, which support the online capacity management approach by implementing a closed MAPE-K control loop.

- Kieker is an extensible framework for application performance monitoring and dynamic software analysis. Kieker enables the technical instrumentation, continuous monitoring, and analysis of software systems.
- SLAstic is an extensible framework for self-adaptive architecture-based online capacity management through architectural runtime reconfiguration. The SLAstic framework provides a separation of architecture and technology: the online analysis is based on architectural runtime models—conforming to the aforementioned modeling languages—and continuous QoS measures; it is possible to connect the framework to systems implemented with different technologies.

For both frameworks, a conceptual architecture is described and a publicly available implementation has been developed.

1.2.3 Model-Driven Online Capacity Management

We developed techniques to improve the automation of reoccurring, schematic tasks within the SLAstic approach employing model-driven techniques. Particularly, this includes *a*) the generation of a Kieker-based instrumentation, *b*) the transformation of low-level monitoring data obtained from Kieker into architecture-level monitoring events, and *c*) the extraction of SLAstic models from monitoring data.

1.2.4 Runtime Reconfiguration for Controlling Capacity

We defined a set of five architectural reconfiguration operations which can be used to control the capacity of a software system and integrated these operations into the SLAstic approach. The set of operations comprises two system-level operations—allocation and deallocation of execution containers (e. g., servers)—and three application-level operations—replication, dereplication, and migration of software components.

1.2.5 Integration of PCM

An orthogonal category of contributions comprises results of using and integrating the Palladio Component Model (PCM) [Becker et al., 2009] into our approach. By providing a transformation from SLAstic instances to PCM instances, it is possible to extract PCM instances by dynamic analysis using the model-driven techniques listed in Section 1.2.3. In order to allow manual refinement/completion of PCM instances and their use at runtime, we developed a decorator concept. We developed the SLAstic.SIM discrete-event simulator for runtime reconfigurable PCM instances, which is integrated with the Kieker and SLAstic frameworks. The simulator includes a PCM-specific implementation of the five proposed runtime reconfiguration operations (Section 1.2.4).

1.2.6 Evaluation

We evaluated our SLAstic approach in a combination of case study, lab experiments, and simulation. In the case study described in this thesis, we deployed our Kieker framework in an industrial case study system and extracted real usage profiles and system models from this data. In lab experiments, we integrated our SLAstic framework with an IaaS cloud infrastructure. A sample Java EE application was deployed to this infrastructure and its capacity was managed by the SLAstic framework. In a simulation-based evaluation, we integrated the framework with the aforementioned simulator and investigated the impact of the approach on system capacity. Moreover, we provide a retrospective view on the history of Kieker's evolution during the past years, including an overview of impact in research and industry.

1.3 Document Structure

The remainder of this document is structured as follows:

- Part I comprises the foundations of this work.
  - Chapter 2 introduces core concepts of model-driven software engineering (MDSE), including modeling languages, model transformations, and MDSE technologies.

- ▻ Chapter 3 introduces the topic of software architectures, focusing on software architecture description, component-based software architectures, technologies for enterprise application systems, as well as core concepts of self-adaptive software systems, including architectural runtime reconfiguration.
- ▻ Chapter 4 introduces the topic of QoS evaluation and capacity management, focusing on QoS measures, SLA languages, as well as performance and workload measurement, modeling, and prediction. The supporting modeling languages, MAMBA and S/T/A, are described in this chapter.

▻ Part II comprises the description of our model-driven online capacity management approach for component-based software systems.

- ▻ Chapter 5 describes our research methodology, including work packages, research questions, and a summary of results.
- ▻ Chapter 6 describes the SLAstic meta-model, including the integration of MAMBA and S/T/A.
- ▻ Chapter 7 gives an overall overview about the Kieker framework, including a brief description of the implementation.
- ▻ Chapter 8 presents the SLAstic framework, including its conceptual architecture and a brief description of its implementation.
- ▻ Chapter 9 describes the model-driven techniques for instrumentation, transformation of monitoring events, as well as model extraction based on dynamic analysis.
- ▻ Chapter 10 focuses on runtime reconfiguration for increasing resource efficiency. It describes the five reconfiguration operations and their integration into the SLAstic modeling languages and framework.
- ▻ Chapter 11 covers the integration of the Palladio Component Model (PCM) into our approach, including the transformation of SLAstic models into PCM instances, the decoration of PCM instances, as well as the simulation of runtime reconfigurable PCM instances supporting the reconfiguration operations described in Chapter 10.

- Part III comprises the evaluation of the approach.
  - Chapter 12 describes the results from the industrial case study.
  - Chapter 14 describes the results using the developed simulation infrastructure.
  - Chapter 13 describes the results from the lab experiments conducted in a private cloud infrastructure.
  - Chapter 15 gives a retrospective overview about the history, development, and impact of the Kieker framework.
  - Chapter 16 discusses related work.
- Part IV draws the conclusions (Chapter 17) and outlines future work (Chapter 18).

The end matter includes lists of acronyms, figures, and tables, as well as the bibliography. Supplementary material for this thesis—comprising software, models, data, etc.—is publicly available online [van Hoorn, 2014].

Part I

Foundations

Chapter 2

Model-Driven Software Engineering

Since decades, models play a core role in various disciplines, including software engineering [Ludewig, 2003; Ludewig and Lichter, 2010]. A commonly used reference for the notion and semantics of a *model* is Stachowiak [1973] who states that a model must meet the three criteria of *a*) *mapping*, *b*) *reduction* (abstraction), and *c*) *pragmatics*; i. e., a model represents a relevant subset (reduction) of a real-world object's (mapping) properties for a specific purpose (pragmatics). Models are typically used to reduce complexity by abstraction and by omitting irrelevant details. Note that the general notion of a model is not limited to graphs or diagrams. The same model may serve for *descriptive* and *prescriptive* purposes.

This chapter introduces core concepts and technologies for model-driven software engineering (MDSE) [Stahl and Völter, 2006; Brambilla et al., 2012], which is an engineering paradigm that handles models and their transformations as primary artifacts to develop, analyze, and evolve software systems. Note that various acronyms with the same or a similar meaning exist, e. g., model-driven development (MDD), model-driven engineering (MDE), and model-driven software development (MDSD). Also, the notion of model-*based* is often used. For this thesis, we chose to select MDSE to include other software engineering activities in addition to *development* as well as to emphasize the focus on software systems and that models serve as primary artifacts—as opposed to model-*based*.

During the past years, MDSE techniques and technologies gained widespread use in academia and industrial practice for different purposes, including generative software development ([Stahl and Völter, 2006]), model-based software performance engineering ([Cortellessa et al., 2011]), as well as reverse and reengineering.

Core concepts of MDSE are modeling languages and model transformations, which are introduced in the following Sections 2.1 and 2.2. Section 2.3 provides an overview of widely used MDSE specifications and technologies contributed by the Object Management Group and the Eclipse Modeling Project. Note that we do not aim to provide a comprehensive introduction to MDSE in this thesis. In order to get deeper into this topic, we recommend the textbooks by Stahl and Völter [2006] as well as by Brambilla et al. [2012].

2.1 Modeling Languages

A modeling language is a formalism to express models. It comprises definitions of *abstract syntax*, *concrete syntax*, and *semantics* [Brambilla et al., 2012]. The abstract syntax specifies the set of modeling primitives along with rules on how to combine them—independent of any representation. Such representations are specified in the concrete syntax, which may be a *textual concrete syntax (TCS)*, a *graphical concrete syntax (GCS)*, or a hybrid combination of both. The meaning of models expressed in the language is provided by the semantics.

In the MDSE context, the abstract syntax of a modeling language is defined by a so-called meta-model, which describes relevant concepts of the problem domain in terms of its entities and their relationships. The formalisms used to express entities and relationships in a meta-model are

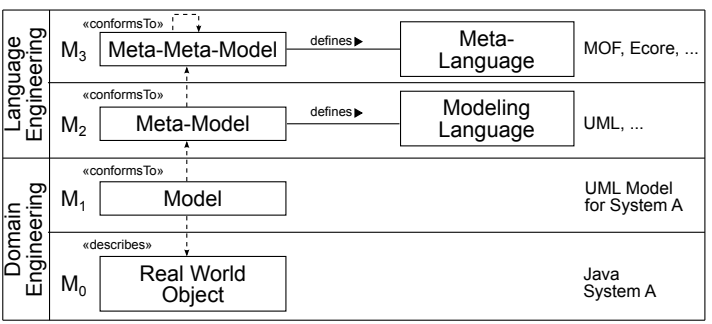


Figure 2.1. Four-layered meta-modeling stack (based on Brambilla et al. [2012])

again specified in a model called meta-meta-model. Similar to concepts found in UML class diagrams [Object Management Group, Inc., 2013b], a meta-meta-model typically provides concepts like (abstract and concrete) classes with typed attributes, class hierarchies through generalization, as well as (un)directed associations among classes, including the containment property.

Figure 2.1 depicts the four-layered stack most MDSE approaches and technologies are based on. Meta-meta-models like MOF or Ecore (see Sections 2.3.1 and 2.3.2)—typically along with additional constraints expressed in Object Constraint Language (OCL) [Object Management Group, Inc., 2012a]—provide the meta-language to express modeling languages defined by a meta-model. These meta-models are used to express models of real-world objects. A common notion is that a model on meta-modeling layer M_i *conforms to* its meta-model on layer M_{i+1} [Kühne, 2006]. The meta-meta-models on layer M_3 are usually self-describing, i. e., they provide their own meta-model.

Modeling languages can be divided into domain-specific languages (DSLs) and general purpose languages (GPLs), depending on whether or not the concepts provided by the language are tailored to a specific problem domain. In many cases, the design and implementation of a DSL is part of an MDSE-based development process.

Distributed and collaborative access to models—on each of the mentioned layers—is supported by so-called *model repositories*. Model repositories can be considered to provide services known from database management systems and version control systems, including persistence, versioning, querying, partial loading, consistence, and transaction management [Kiel, 2013].

2.2 Model Transformations

A model transformation is a program that takes a set of *source models* as input and maps these to a set of *target models* produced as output. Each source and target model conforms to a meta-model. Figure 2.2 depicts the general schema of model transformations. An established model transformation taxonomy exists [Mens and Van Gorp, 2006], which classifies transformations based on orthogonal dimensions, such as *a*) the number

of involved source and target models—*one-to-one*, *one-to-many*, *many-to-one*, *b*) whether source and target models conform to the same or different meta-models—*endogenous* vs. *exogenous*, *c*) whether source and target models are on the same or different levels of abstraction—*horizontal* (*refactoring*, *migration*) vs. *vertical* (*abstraction* vs. *refinement*), *d*) and whether the source and target models are the same or different models—*in-place* vs. *out-place*. Transformations whose source or target models are expressed in a TCS (e. g., source code)—typically without an explicit meta-model—are referred to as model-to-text (M2T) and text-to-model (T2M) transformations. Otherwise, transformation are called model-to-model (M2M). M2T transformations are also called *generators*.

Transformations are expressed using formalisms called *transformation languages*. Typically, the exact four-layered meta-model stack described in Section 2.1 is the basis for transformations (M_1) and transformation languages (M_2), i. e., a transformation is a model that conforms to a meta-model defining the transformation language (Figure 2.2). This allows transformations to be used as input and/or output for transformations, which are then called higher order transformations (HOTs). A number of transformation languages with corresponding tool support exist, which differ quite considerably, e. g., in terms of the supported transformation dimensions (see above) and meta-meta-models, as well as language characteristics and underlying transformation approaches. With respect to the latter aspects, M2M transformations may, for instance, be based on direct manipulation, relational (declarative) or operational (imperative) constructs, graph grammars, as well as combinations of these [Czarnecki and Helsen, 2006].

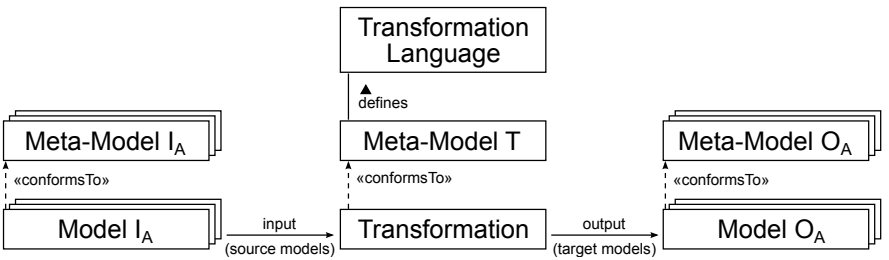


Figure 2.2. Model transformation schema

2.3 Technologies

Sections 2.3.1 and 2.3.2 describe MDSE specifications by the Object Management Group and the Eclipse Modeling Project, focusing on those parts that are relevant to this thesis.

2.3.1 Specifications by the Object Management Group

Under the umbrella of the Model-Driven Architecture (MDA) [Object Management Group, Inc., 2003, 2013d] framework, the Object Management Group (OMG) is maintaining a number of specifications that emerged as de-facto standards for MDSE approaches and tools today. In this section, we will briefly mention the purpose of the most important MDA specifications that are relevant to this thesis. We also include a modeling specification that is being developed as part of the OMG's Architecture-Driven Modernization (ADM) initiative [Object Management Group, Inc., 2013c]. We chose not to list the long names, latest version numbers and references for the specification in the text. This information is provided in Table 2.1.

MOF (including CMOF and EMOF) is the self-describing meta-meta-model used by all other OMG modeling specifications. OCL is a side-effect-free declarative language that allows to refine MOF-based models by constraints, invariants, queries, etc. XMI and HUTN specify an XML-based interchange format and a standard TCS for MOF-based models. QVT specifies one imperative and two declarative model transformation languages. UML, which is probably the most famous OMG specification, is an architecture description language (ADL) (Section 3.1.4) that allows to describe structural and behavioral aspects of software-intensive systems. The UML can also be used to define DSLs, by creating so-called UML Profiles. Relevant to this thesis are the UML Profiles SPT and its successor MARTE (covered in Section 4.5), which allow to enrich UML diagrams by performance-relevant information. SMM, which will be detailed in Section 4.1.4, provides a meta-model to formulate measures and measurements relating to MOF-based models.

Table 2.1. Important OMG modeling specifications, divided into generic meta-modeling and transformation specifications, modeling languages, and UML Profiles.

Acronym	**Specification name** ***[Reference incl. year]***	**Version**
HUTN	Human-Usable Textual Notation *[Object Management Group, Inc., 2004]*	1.0
MOF	Meta Object Facility *[Object Management Group, Inc., 2011a]*	2.4.1
OCL	Object Constraint Language *[Object Management Group, Inc., 2012a]*	2.3.1
QVT	Query/View/Transformation *[Object Management Group, Inc., 2011b]*	1.1
XMI	XML Metadata Interchange *[Object Management Group, Inc., 2013a]*	2.4.1
SMM	Structured Metrics Meta-Model *[Object Management Group, Inc., 2012b]*	1.0
UML	Unified Modeling Language *[Object Management Group, Inc., 2013b]*	2.5
MARTE	UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems *[Object Management Group, Inc., 2011c]*	1.1
SPT	UML Profile for Schedulability, Performance, and Time *[Object Management Group, Inc., 2005]*	1.1

2.3.2 Eclipse Modeling Project

The Eclipse Modeling Project (EMP) [Eclipse Foundation, 2014] provides MDSE technologies that are widely used in research and practice. The EMP technologies, organized in EMP subprojects, can be grouped into support for abstract and concrete syntax development, model transformation, and implementations of industry-standard meta-models. In this section, we will briefly mention EMP subprojects relevant to this thesis. We decided not to include URLs for the web sites of the every EMP subproject, as they are reachable via the main EMP web site [Eclipse Foundation, 2014].

▻ *Abstract and concrete syntax development.*

The most prominent part of EMP is the (meta-)modeling and code generation framework EMF (Eclipse Modeling Framework) [Steinberg et al., 2009], which provides the basis for all other EMP technologies. It includes the self-describing meta-meta-model Ecore, which is almost equivalent to the OMG's EMOF. Other EMP projects contribute additional functionality for EMF, e. g., for querying, validation, and transaction management. The Connected Data Objects (CDO) subproject provides a repository for distributed shared EMF (meta-)models, with properties like persistence, concurrent and transactional access, collaboration, and fail over.

With respect to concrete syntax development, EMP subprojects support the development of graphical editors, as well as textual syntaxes and editors. With respect to the latter, Xtext is a popular framework for developing TCSs and programming languages [Efftinge et al., 2012].

▻ *Transformation.*

M2M and M2T transformation languages and tools provided by EMP subprojects include ATL, Henshin, and Xtend. The latter builds on the afore-mentioned Xtext technology.

▻ *Support for industry-standard meta-models.*

Ecore-based implementations of industry-standard meta-models are provided, e. g., the OMG's OCL, UML 2 (both mentioned in Section 2.3.1), and BPMN2 [Object Management Group, Inc., 2011d] specifications. In addition to this, EMP subprojects in this group provide tool support. Examples include *Papyrus*, which is a graphical UML 2 editor, and *MoDisco*, which provides tool support for the OMG's MDA initiative (Section 2.3.1), including model extractors and transformations.

Additional details about EMF and ATL will be provided in the following two sections, as they are used as (meta-)modeling and M2M transformation technologies in this thesis.

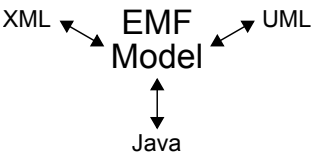


Figure 2.3. Equivalent representations for EMF models [Steinberg et al., 2009]

Eclipse Modeling Framework

As mentioned before, EMF includes the self-describing meta-meta-model (M_3 layer, Section 2.1) called Ecore, which—similar to MOF (Section 2.3.1)—can be seen as the class diagram subset of UML. Meta-models (M_2) that conform to Ecore are referred to as *EMF models*. Three equivalent representations for EMF models (M_2) exist: *a*) XML schemas, *b*) UML class diagrams, and *c*) Java programs. As depicted in Figure 2.3, EMF supports to transform each representation into any of the other representations. The XML schema conforms to an Ecore-compatible variant of XMI, called *Ecore XMI*, with the file extension *.ecore*. Model instances (M_1) can be serialized to XMI, conforming to the XML schema representing the EMF model. Many UML tools provide an export option to *Ecore XMI*. EMF includes a configurable code generator that transforms EMF models into Java programs, including special annotations enabling manual refinements in the code.

Ecore's core meta-classes are `EClass`, `EDataType`, `EAttribute`, and `EReference`. An `EClass`, which is the meta-class for classes in conforming EMF models, has a name, refers to zero or more supertypes (`EClass`), may be marked as an abstract class or an interface, and contains two types of so-called structural features—`EAttributes`, `EDataType`, and `EReferences`, which are associations to other `EClasses`. Concrete `EDataTypes` are included for common plain types like integers (`EInt`) and boolean (`EBoolean`). Note that Ecore contains considerably more meta-classes and associations. Please refer to Steinberg et al. [2009] for a detailed Ecore description.

Usually, the UML class diagram notation is used to visualize the abstract syntax of EMF models. Steinberg et al. [2009] elaborate this mapping in detail. Likewise, UML Object Diagrams are used to visualize model instances (M_1).

EMF provides various additional features not relevant to this thesis, e. g., validation, persistence, reflective API, runtime framework, editing in a UI, etc. Please refer to Steinberg et al. [2009] for a comprehensive introduction to EMF.

ATLAS Transformation Language

The ATLAS Transformation Language (ATL) [Jouault et al., 2008] is a M2M transformation language and a corresponding tool set, developed as part of the EMP. An ATL transformation program produces a set of target models from a set of source models. An ATL program is composed of rules that define how to create target model elements from source model elements. ATL is a hybrid transformation language, i. e., it supports the use of both declarative and imperative styles in defining the transformation rules. The ATL tool set allows to edit, compile, launch, and debug ATL transformation programs. To this date, the ATL transformation engine supports three meta-meta-model technologies:[1] *a*) MOF (see Section 2.3.1), *b*) Ecore, and *c*) KM3 [Jouault and Bézivin, 2006]. Details on ATL are described by Jouault et al. [2008] and have also been elaborated by Günther [2011] in the context of this thesis.

[1] `http://wiki.eclipse.org/ATL/Concepts`

Chapter 3

Software System Architecture: Description and Reconfiguration

Each software system exhibits a so-called *architecture*, informally comprising its fundamental properties and design decisions, e. g., with respect to structure and behavior [Taylor et al., 2009]. Architectural descriptions [ISO/IEC/IEEE, 2011] serve to make certain aspects of an architecture explicit for specific purposes, e. g., for documenting the set of system components and their interactions. Common and reusable concepts for the description of software system architectures emerged and provide the foundation for architecture-based approaches for system evaluation and reconfiguration.

This chapter introduces relevant concepts for architecture description and reconfiguration, with a focus on component-based software systems. It is organized as follows. Section 3.1 introduces common terminology and concepts used to describe software system architectures. Section 3.2 focuses on architectures of component-based software systems. Technologies for enterprise application systems are introduced in Section 3.3, focusing on Java and IaaS-based cloud computing. Foundations on self-adaptive software systems, including architectural runtime reconfiguration, are provided in Section 3.4.

3.1 Describing Software System Architectures

A whole body of work on architectures of software(-intensive) systems has been published since the early 1990s, when this became a hot research topic [Kruchten et al., 2006]. Numerous definitions for the term architecture have been proposed—see, for example, the collection of definitions maintained

by the Carnegie Mellon Software Engineering Institute (SEI) in its online software architecture glossary.[1] In this thesis, we use Definition 3.1 from the ISO/IEC/IEEE International Standard 42010:2011(E) [ISO/IEC/IEEE, 2011]. The standard makes no specific assumptions about the kind of system whose architecture is to be described. However, it explicitly includes the domain of *software-intensive systems*, as defined in the IEEE Standard 1471-2000 [IEEE, 2000].

Definition 3.1 ((System) architecture [ISO/IEC/IEEE, 2011])**.** *"Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution."*

The following sections introduce terms, definitions, and concepts from software architecture research, which are relevant to this thesis: *a*) architecture description in the afore-mentioned ISO/IEC/IEEE standard (Section 3.1.1), *b*) component, connector, and configuration (Section 3.1.2), *c*) architectural style (Section 3.1.3), *d*) and architecture description languages (Section 3.1.4). For further details, comprehensive textbooks by researchers having significant impact on advances in this area exist—for example, by Clements et al. [2002] and Taylor et al. [2009].

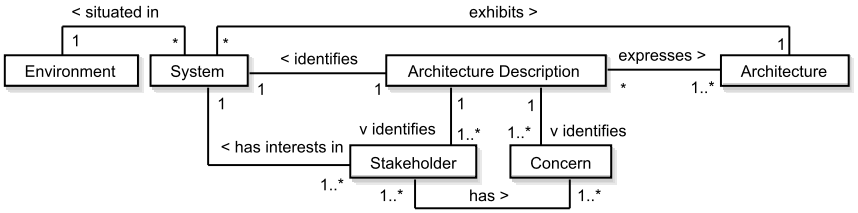
3.1.1 Architecture Description in ISO/IEC/IEEE Std. 42010

The ISO/IEC/IEEE International Standard 42010:2011(E) (*Systems and Software Engineering — Architecture Description*) [ISO/IEC/IEEE, 2011], which is the most recent successor of the widely known IEEE Standard 1471-2000 (*IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*) [IEEE, 2000], suggests a conceptual model of *architecture description* for *systems* of interest. *Architecture description*, which is the central notion in the standard, is defined as follows (Definition 3.2):

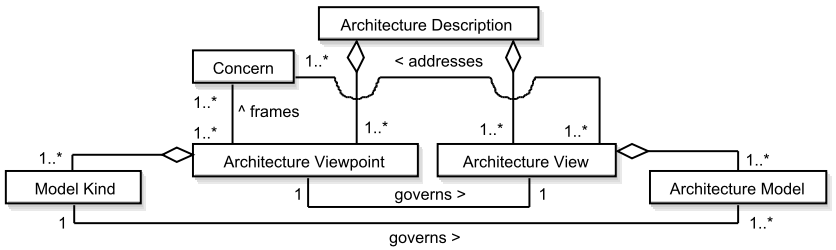
Definition 3.2 (Architecture description [ISO/IEC/IEEE, 2011])**.** *"Work product used to express an architecture."*

Limited to those parts relevant to this thesis, Figure 3.1 depicts the context and the conceptual model of an architecture description as defined by the standard. The important drivers for creating an architecture description are

[1] `http://www.sei.cmu.edu/architecture/start/glossary`



(a) Context of architecture description



(b) Conceptual model of an architecture description (excerpt)

Figure 3.1. Context (a) and conceptual model (b) of an architecture description according to the ISO/IEC/IEEE Standard 42010:2011(E) [ISO/IEC/IEEE, 2011]

concerns relevant to one or more *stakeholders* having interests in the system (see also Figure 3.1a). Note that those concerns—just like the environment the system is situated in—includes but is not limited to technical aspects like functional or non-functional requirements. The concerns are addressed by *architecture views*, contained in the architecture description, which are expressed by one or more *architecture models*. As depicted in Figure 3.1b, an *architecture view* (view) and an *architecture model* are governed by an *architecture viewpoint* (viewpoint) and a *model kind* respectively. Definitions 3.3 to 3.5 list definitions for the previously mentioned terms as included in the standard.

Definition 3.3 ((Architecture) view [ISO/IEC/IEEE, 2011])**.** *"Work product expressing the architecture of a system from the perspective of specific system concerns."*

Definition 3.4 ((Architecture) viewpoint [ISO/IEC/IEEE, 2011])**.** *"Work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns."*

Definition 3.5 (Model kind [ISO/IEC/IEEE, 2011])**.** *"Conventions for a type of modelling."*

For example, let's assume that a customer (stakeholder) requests a response time guarantee for a system-provided service (concern). As will be described in Section 4.5, a number of performance prediction techniques exist. To frame the afore-mentioned concern, a viewpoint could specify that a Layered Queueing Network (LQN), as the model kind, is to be used to predict the measure of interest based on a specific LQN solver configuration. The concrete analysis for the system, based on an LQN model (architecture model), would represent the view addressing the afore-mentioned concern.

It makes sense to sort some of the afore-mentioned concepts for architecture description into the four-layer meta-modeling stack from MDSE (see Section 2.1)—even though MDSE technology is not necessarily employed. Environment, system, architecture, stakeholders, and concerns (Figure 3.1a) are real-world entities situated on the M_0 layer. In order to define the concepts (M_2) to be used for architecture description, a set of architecture viewpoints are selected, which imply the set of used model kinds. Countless existing notations on different levels of abstraction may serve as model kinds, e. g., design-level software modeling languages like the UML [Object Management Group, Inc., 2013b] or more generic and abstract languages like Petri Nets or Queueing Networks (Section 4.5). The notion of architecture description languages (ADLs), detailed in Section 3.1.4, is often used for these modeling languages in the context of architecture description. A set of architecture models (grouped in architecture views), located on the M_1 layer, conform to the previously mentioned concepts on the $M2$ layer and describe the real-world entities on M_0.

Note that just like any model (page 11), architecture descriptions may be prescriptive ("as-designed", "as-intended") or descriptive ("as-implemented", "as-realized"). See Taylor et al. [2009]) for a discussion on this.

3.1.2 Component, Connector, and Configuration

Definition 3.1 includes the notion of architectural *elements* and *relationships*. In the literature, these elements or usually divided into *components* and *connectors* that can be made of *software* or *hardware*. With respect to software, Shaw and Clements [1997] define a component as "a unit of software, that performs some function at runtime" and list objects and processes as examples. (In Section 3.2, we will refine our understanding of a software component for this thesis.) With respect to hardware components, these may include server nodes, network routers, or lower-level resources like CPUs. Connectors mediate interaction among components and often have no corresponding element in the real system [Shaw and Clements, 1997]. Example software connectors include mechanisms for local or remote procedure calls, or event-based communication. A typical example of a relationship type is the information to which hardware component(s) a software component is deployed to. Both the sets of *elements* and *relationships* in an architecture may change as part of a system's evolution, including changes at runtime as part of runtime reconfiguration (Section 3.4.2).

A common notion for an architectural snapshot in terms of *elements* and *relationships* is the *architectural configuration*. Fielding [2000] contributes the appropriate Definition 3.6:

Definition 3.6 (Architectural Configuration [Fielding, 2000])**.** *"A configuration is the structure of architectural relationships among components, connectors, and data during a period of system runtime."*

3.1.3 Architectural Style

A common and useful notion to describe high-level architectural decisions shared by multiple architectures is *architectural style*. Taylor et al. [2009] provide an informal definition (Definition 3.7):

Definition 3.7 (Architectural style [Taylor et al., 2009])**.** *"An architectural style is a named collection of architectural design decisions that a) are applicable in a given development context, b) constrain architectural design decisions that are specific to a particular system within that context, and c) elicit beneficial qualities in each resulting system."*

A more technical definition, using the terminology introduced in this chapter, is given by Fielding [2000] (Definition 3.8):

Definition 3.8 (Architectural style [Fielding, 2000])**.** *"An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style."*

Example architectural styles are client-server, pipes-and-filters, event-based systems, service-oriented architecture (SOA), and Representational State Transfer (REST) [Fielding and Taylor, 2002]. Taylor et al. [2009] describe a large number of common architectural styles using a common description template. This thesis also builds on a wide-spread architectural style, namely component-based software systems, as detailed in Section 3.2.

Similar to design patterns, architectural patterns, and reference architectures, architectural styles typically result from solutions that have proven useful and—as a way of reuse—serve as blueprints for solving similar problems. Also, architectural style enables to develop architectural analyses that can be applied to any system that conforms to a respective style, e. g., performance predictions for component-based software systems (Section 4.5).

3.1.4 Architecture Description Languages

Section 3.1.1 introduced the context and the conceptual model of architecture description as defined in the ISO/IEC/IEEE Standard 42010. In that section, we've already mentioned that formalisms are needed to express architecture views and models that conform to architecture viewpoints and model kinds. These formalisms are provided by architecture description languages (ADLs). An ADL definition that uses the terminology introduced before is provided by Clements et al. [2002] (Definition 3.9):

Definition 3.9 (Architecture description language [Clements et al., 2002])**.** *"A language (graphical, textual, or both) for describing a software system in terms of its architectural elements and the relationships among them."*

Based on Section 3.1.2, note that the typical types of architectural elements to be supported by an ADL are components, connectors, and configurations [Medvidovic and Taylor, 2000].

In the context of this thesis, we assume that an ADL is a modeling language that is formal in the following sense: it is or can be defined by a meta-model on the M_2 layer of the four-layered MDSE meta-modeling stack introduced in Section 2.1. However, an explicit concrete syntax is not required. This allows to use architecture descriptions in tool-supported MDSE contexts, e. g., for model analysis and transformations.

Note that these strict assumptions on ADLs are not always required; Taylor et al. [2009], for instance, have a rather broad view on an ADL's requirements: "ADLs can be textual or graphical, informal (such as PowerPoint diagrams), semi-formal, or formal, domain-specific or general-purpose, proprietary or standardized, and so on."

The requirement for tool support is also confirmed by Medvidovic and Taylor [2000] in their frequently cited article on classifying and comparing ADLs. State-of-the-art MDSE technology eases the development of ADLs, including TCSs and GCSs as well as corresponding tool support.

A comprehensive description of (the history of) ADLs is provided by Taylor et al. [2009]. The authors classify the covered ADLs into three categories: *a*) early, first generation ADLs from research projects that are no longer active (e. g., Darwin [Magee et al., 1995], Rapide [Luckham and Vera, 1995], and and Wright [Allen and Garlan, 1997]), *b*) domain- and style-specific ADLs (e. g., Koala [van Ommering et al., 2000], Weaves [Gorlick and Razouk, 1991], and Architecture Analysis and Design Language (AADL) [Feiler et al., 2003]), *c*) and extensible ADLs (e. g., Acme [Garlan et al., 1997] and xADL [Dashofy et al., 2005]).

ADLs are sometimes tailored for specific architectural styles and purposes. Of particular interest to this thesis are ADLs to describe component-based software architectures (CBSAs), as detailed in Section 3.2. We focus on the existing ADLs UML 2 and PCM and develop a custom ADL for CBSAs in Chapter 6.

3.2 Component-Based Software Architectures

In this thesis, we focus on so-called *component-based software systems (CBSSs)*, i. e., software systems that are created by the assembly of *software components*. The idea of CBSE was first proposed by Mcilroy [1969] in 1968 at a NATO software engineering conference. The motivation—based on common prac-

tice in other engineering disciplines—has been to have a market for reusable and well-specified software components that can be reused for different CBSSs. As generally accepted in software engineering, reuse may increase the quality of a software product and decrease development costs.

The term *component* in this context is meant to be more specific compared to the one previously introduced in Section 3.1.2—i. e., simply being one type of architectural element from Definition 3.1. A common definition for *software components* accepted in the research community emerged in 1996 during the Workshop on Component-Oriented Programming (WCOP):

Definition 3.10 (Software component [Szyperski et al., 2002])**.** *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

While this definition provides a general understanding of the term and fits to the previous description of CBSSs, it is quite unspecific about how a software component looks like. There have been lots of debates about a definition of what a software component is and what it is not—which usually depends quite a lot on the domain and the properties of respective CBSS, e. g., business-critical enterprise application systems as apposed to safety-critical embedded systems. Heineman and Councill [2001] include the notion of a component model (Definition 3.12) into their definition (Definition 3.11):

Definition 3.11 (Software component [Heineman and Councill, 2001])**.** *"A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."*

Definition 3.12 (Component model [Heineman and Councill, 2001])**.** *"A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment."*

A number of component models have been developed during the past years for several domains, technologies, and purposes. A recent overview, along with a classification framework, is provided by Crnković et al. [2011]. Prominent component models from academia and research include CORBA Component Model (CCM) [Object Management Group, Inc., 2006],

COM [Box, 1998], Enterprise JavaBeans (EJB) (Section 3.3.1), OSGi [OSGi Alliance, 2012], Koala [van Ommering et al., 2000] (already mentioned in Section 3.1.4), Fractal [Bruneton et al., 2006], SOFA [Bureš et al., 2006], as well as the Palladio Component Model (PCM) [Becker et al., 2009] with a focus on design-time performance prediction. Together with other component models, the latter three have also been applied to the Common Component Modeling Example (CoCoME) [Rausch et al., 2008]. Component models often include an ADL (Section 3.1.4) and induce a certain (component-based) architectural style (Section 3.1.3) to systems using the component model.

The component model assumed in this thesis builds on the aforementioned PCM, which will be detailed in Section 4.5. Our core assumptions w.r.t. the component model and the architectural style are: *a*) a system-independent repository provides descriptions of self-contained and reusable software components (and other component types), including the definition of provided and required interfaces; *b*) a system's logical software architecture is described by the selection of software components from the repository and assembling them via connectors; *c*) each of the system's software components is deployed to one or more execution containers (e. g., physical servers) allocated for the system. Note that the UML 2 provides capabilities to describe and visualize CBSAs, as detailed in the respective part of the specification [Object Management Group, Inc., 2013b]. We use UML 2-like visualization as a GCS for our SLAstic meta-model described in Chapter 6.

3.3 Enterprise Application System Technologies

The approach developed in this thesis is architecture-based, i. e., it makes no specific assumptions about what technologies are used to implement the monitored and controlled system. However, note that we focus on distributed enterprise application systems (EASs). EASs are business-critical software systems, such as online shopping sites and customer portals, as well as customer relationship management and enterprise resource planning systems. Sections 3.3.1 and 3.3.2 briefly introduce state-of-the-art Java and cloud technologies for implementing such systems. Particularly, this includes those technologies that occur as part of the proof-of-concept implementations (e. g., Section 7.3) and the evaluation in case studies and lab

Table 3.1. Selected Java SE and Java EE technologies. Additional information is available on the Java SE/Java EE web sites [Oracle, 2014c,b] and the respective JSRs.

Short	Long name	JSR	Latest version	Year
EJB	Enterprise JavaBeans	345	3.2	2013
Java EE	Java Platform, Enterprise Edition	342	7	2013
JMS	Java Message Service	343	2.0	2013
JMX	Java Management Extensions	003	1.2	2006
JNDI	Java Naming and Directory Interface	—	1.2.1	1999
JSF	JavaServer Faces	344	2.2	2013
JSP	JavaServer Pages	245	2.1	2013
JPA	Java Java Persistence API	338	2.1	2013
JTA	Java Transaction API	907	1.2	2013
Servlet	Java Servlet	340	3.1	2013

experiments (Chapters 12 and 13). For an introduction on the underlying concepts, principles, and patterns for implementing such systems—e. g., application layering, distributed web-based systems, middleware, communication, including remote procedure call (RPC) and message-oriented middleware (MOM)—refer to respective textbooks like the ones by Fowler [2002] and Tanenbaum and van Steen [2008].

3.3.1 Java Technologies

During the past decade, Java emerged to a widely used programming language and platform for implementing EASs. The Java Platform, Standard Edition (Java SE) already includes a number of libraries and APIs for implementing EASs, e. g., for UIs (Swing etc.), database connectivity (JDBC), networking and RPC (RMI etc.), monitoring and management (JVMTI, JMX, etc.), and naming (JNDI) [Oracle, 2014c]. The Java Platform, Enterprise Edition (Java EE) adds further technologies for EASs, particularly for web-based and multi-tiered architectures [Oracle, 2014b]. Core Java EE technologies for the web tier include Servlet, JavaServer Pages (JSP), and JavaServer Faces (JSF). The implementation of the business tier is supported by the Enterprise JavaBeans (EJB) component technology, including support for naming and directory services (based on JNDI), RPC (based on RMI

and web services compatible with SOAP [World Wide Web Consortium (W3C), 2007a], WSDL [World Wide Web Consortium (W3C), 2007b], etc.), transaction management (based on JTA), messaging (based on JMS), and persistence (based on JPA). While Java SE applications solely rely only on a JVM for being executed, Java EE applications need to be deployed to an Application Server (AS), which manages the execution of the application. A number of commercial and open source ASs exist, which provide an implementation of all or a subset of the Java EE specifications.

Specifications for most Java SE and Java EE technologies have been and are being developed as so-called Java Specification Requests (JSRs) by Sun Microsystems, Inc. (Sun), Oracle Corporation (Oracle), and the Java Community Process (JCP).[2] Table 3.1 lists a selection of the mentioned technologies with the latest version number and year of its publication, as well as the JSR document number.

3.3.2 Cloud Computing Technologies

A widely used definitions for cloud computing has been contributed by the National Institute of Standards and Technology (NIST) (Definition 3.13):

Definition 3.13 (Cloud computing [Mell and Grance, 2011])**.** *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e. g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."*

Especially the three service models—software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS)—and the four deployment models—private, community, public, and hybrid cloud— mentioned in Definition 3.13 and detailed by Mell and Grance [2011] became the de-facto criteria to classify clouds. We will not give a deeper introduction into the general concepts of cloud computing and how this can be distinguished from previous computing models. For this, please refer to other resources, e. g., by Armbrust et al. [2009].

[2] http://www.jcp.org/

With respect to the three mentioned service models, IaaS is the appropriate abstraction layer for integration with our SLAstic framework. Typical offerings included in IaaS products include dynamic provisioning of computing, and network resources, typically implemented based on virtualization technology. Various commercial and open-source IaaS products evolved during the last couple of years. The most prominent commercial IaaS products are Amazon Web Services (AWS) [Amazon Web Services, Inc., 2014] and Microsoft's Windows Azure [Microsoft, Inc., 2014], both of which are offered as public clouds on a pay-per-use basis. Popular open source IaaS products include Eucalyptus [Nurmi et al., 2009; Eucalyptus Systems, Inc., 2014] and OpenStack [OpenStack Foundation, 2014], which are both compatible with AWS and may be used to realize private or hybrid clouds. In the remainder of this section, we will briefly describe AWS and Eucalyptus, as these technologies are used in this thesis.

Amazon Web Services

AWS comprises a comprehensive set of cloud services, grouped into the following categories [Amazon Web Services, Inc., 2014]: *a*) compute (e. g., EC2, auto scaling, and elastic load balancing) and networking (e. g., DNS), *b*) storage (e. g., S3 and EBS) and CDN, *c*) database (e. g., RDS), *d*) application services (e. g., MOM and email), *e*) as well as deployment and management (e. g., web-based UI and CloudWatch). The most prominent AWS service is Elastic Compute Cloud (EC2) allowing to dynamically allocate and release virtual machine instances, which are configured by selecting an Amazon Machine Image (AMI)—including an operating system and a software stack—and an instance type—defining the hardware resources allocated for the instance (CPU, memory, etc.). The CloudWatch service allows to collect and visualize measures about AWS resources (e. g., usage of memory, disk, and CPU) and services (e. g., request counts and latencies for the load balancer, and the number of messages in the MOM), as well as custom ones. CloudWatch is the basis for the AWS's auto scaling, which automatically allocates and releases EC2 instances according to rules defined on the CloudWatch statistics. The AWS services can be managed via a web-based UI, a web service API, a CLI, or a Java API. For details on the AWS services and pricing model, please refer to Amazon Web Services, Inc. [2014].

Eucalyptus

Eucalyptus [Eucalyptus Systems, Inc., 2014] is an open source cloud platform, which has been initially started by researchers at the University of California, Santa Barbara. Eucalyptus includes cloud services compatible to the API of the AWS services EC2, EBS, S3, auto scaling, elastic load balancing, and CloudWatch. Eucalyptus Machine Images (EMIs) are compatible to AMIs. The *Euca2ools* provides a set of CLI tools to manage a Eucalyptus cloud. Due to the API compatibility, the afore-mentioned AWS tools can be used as well. For details on Eucalyptus, please refer to Eucalyptus Systems, Inc. [2014].

3.4 Self-Adaptive Software Systems

At the beginning of this century, the direction of self-adaptive systems has been pushed by IBM by launching the *autonomic computing initiative*.[3] A common definition for the term autonomic computing is provided by Kephart and Chess (Definition 3.14):

Definition 3.14 (Autonomic computing [Kephart and Chess, 2003])**.** *"Computing systems that can manage themselves given high-level objectives from administrators."*

The term autonomic computing is mainly used in the systems community with the International Conference on Autonomic Computing (ICAC) being the leading conference in this field.[4] In the software engineering and software architecture research communities, the terms *self-adaptive software* or *self-adaptive software systems (SASSs)* are more frequently used. For these communities, the Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) is the leading event on this topic. A definition from this community is provided by Oreizy et al. (Definition 3.15):

Definition 3.15 (Self-adaptive software [Oreizy et al., 1999])**.** *"Self-adaptive software modifies its own behavior in response to changes in its operating environment."*

[3] http://www.research.ibm.com/autonomic/

[4] http://www.autonomic-conference.org/

In the remainder of this section, we will briefly cover SASS architectures including the MAPE-K control loop (Section 3.4.1), architectural and architecture-based adaptation (Section 3.4.2), as well as the S/T/A modeling approach for adaptation processes developed in the context of this thesis (Section 3.4.3). For a deeper introduction on SASSs, we refer to Kramer and Magee [2007], Huebscher and McCann [2008], Salehie and Tahvildari [2009], Cheng et al. [2009], and de Lemos et al. [2013], who provide state-of-the art summaries in this area and highlight research challenges. Taylor et al. [2009] may be used as an introduction to architecture-based approaches to SASSs.

3.4.1 SASS Architecture and MAPE-K Control Loop

Architectures of self-adaptive software systems (SASSs) comprise the *adaptable software system*, which is subject to adaptation, as well as the *adaptation engine*, which decides when and how to adapt the software system. Communication between both parts is performed via *sensors* and *effectors*, which are provided by the adaptable software system. Sensors—also referred to as *monitors*—collect and report runtime observations from the adaptable software system, providing information about the system's current state to the adaptation engine. Such observations include performance measurements like CPU utilization, memory usage, workload intensity, and response times of software operations. Techniques for performance measurement will be detailed in Section 4.2. Effectors conduct the system adaptations requested by the adaptation engine. Kephart and Chess [2003] propose a reference architecture for SASSs, including the widely known MAPE-K control loop. MAPE-K stands for the monitoring, analysis, planning, and execution processes conducted by the adaptation manager, as well as the common knowledge that builds the foundation for these processes. Note that instead of the terms adaptable software system and adaptation engine, the terms *managed element* and *autonomic manager* are also used in this context. The monitoring process obtains and pre-processes the data received from the sensors—e. g., by filtering or aggregation—and updates the knowledge about the current system state. The analysis decides whether adaptation actions are needed, which are then planned by the adaptation process. The decision making and planning is based on *policies*, e. g., based on rule sets, goals, or utility functions [Kephart and Walsh, 2004]. The

execution process is responsible for managing the adaptation by interacting with the effectors.

3.4.2 Architectural and Architecture-Based Adaptation

The previous introduction to SASSs has made no explicit statements about how the knowledge kept by the adaptation manager looks like. Obviously, this may include technical details about the implementation of the adaptable software system. On the other hand, architecture-based SASSs approaches employ architectural descriptions of the adaptable system—typically expressed using an architecture description language (ADL) (Section 3.1.4)—to decide, plan, and execute adaptations. The remainder of this section introduces core concepts on architectural and architecture-based adaptation.

Supported architectural adaptations are typically defined based on the elements and relationships captured by the ADL. Particularly, this includes components, connectors, and configurations, as described in Section 3.1.2. Hence, example architectural adaptations are the replacement or modification of components and connectors, and the modification of their relationships. Hofmeister [1993] classifies reconfiguration into the levels of *a*) implementation, *b*) geometry, and *c*) structure. Some architectural styles (introduced in Section 3.1.3) explicitly include concepts for architectural adaptation, also referred to as *architectural reconfiguration*.

Of particular interest to this thesis is the execution of architectural adaptation at runtime—referred to as *architectural online adaptation* or *online reconfiguration*. The notion of *architecture-based (online) adaptation* is commonly used to emphasize that along with the architectural description, the real system represented by the architectural model needs to be modified, and vice versa. According to our notion above, in SASSs this is the responsibility of the adaptation engine propagating the desired change via the effectors and keeping a consistent knowledge about the system's architecture. Oreizy et al. [1998, 1999, 2008] proposed a conceptual framework for architecture-based adaptation, including the aspect of maintaining consistency between the architectural model and the implementation. Important results on architecture-based approaches to SASSs have also been contributed by Garlan et al. [2003], e. g., by introducing the concept of style-based adaptation.

Executing architectural adaptations at runtime imposes a number of conceptual and technical challenges as the system is being modified while it is used and hence is in a specific state. Important results in this area, e. g., on transactional change, have been contributed by Kramer and Magee [1985, 1990] under the notion of *dynamic (re)configuration* and *dynamic change management*. Building on this, Matevska [2009] proposed a concept for runtime reconfiguration of component-based software systems.

3.4.3 Modeling Runtime Adaptation Processes with S/T/A

As a result of joint work in the context of this thesis (Section 5.3), we created the S/T/A (strategies/tactics/actions) modeling language that allows to express adaptation processes—involving strategies, tactics, and actions—on an architectural level. A respective interpreter exists that executes the modeled processes. The remainder of this section briefly introduces the core modeling concepts relevant to this thesis. For a comprehensive introduction to S/T/A, please refer to our publication on S/T/A [Huber et al., 2014]. This section includes contents from this publication.

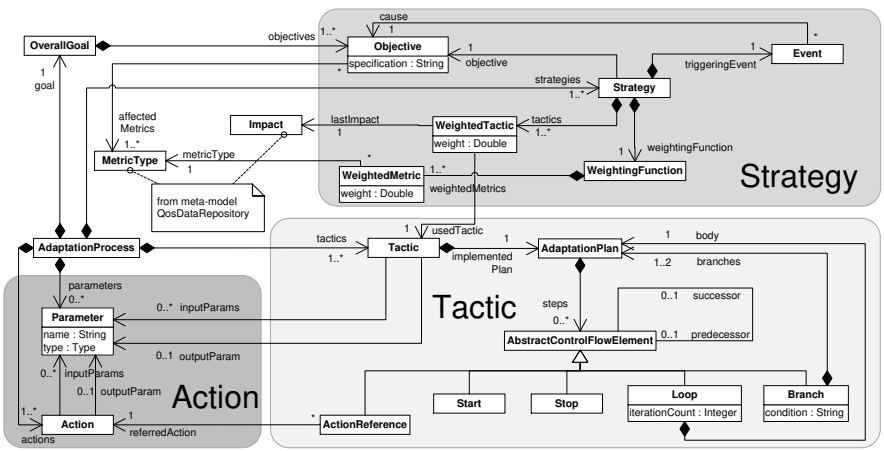


Figure 3.2. S/T/A meta-model [Huber et al., 2014]

According to the previously described architecture-based approach for (self-)adaptation, S/T/A distinguishes between an architectural model of the adaptable system and its implementation. In addition to architectural information about the system, the architectural model includes the definition of allowed adaptation operations, denoted as *degrees of freedom*. S/T/A is ADL-agnostic, i. e., not tailored to a specific architectural modeling formalism. S/T/A provides the modeling formalism to express adaptation processes for the adaptable system as a hierarchy of *objectives*, *strategies*, *tactics*, and *actions*. These adaptation processes are evaluated on the architectural level, i. e., based on the architectural knowledge about the adaptable software system. Architectural runtime reconfigurations (actions) are then executed to the managed system via effectors and system-specific connectors to the effectors.

Figure 3.2 depicts the S/T/A meta-model, including the classes and relationships for the core language concepts, namely *Strategy*, *Tactic*, and *Action*. Each strategy (meta-class `Strategy`) aims to achieve a given high-level objective (`Objective`), e. g., meeting performance requirements specified as part of the service level agreements (Section 4.1.3). A strategy uses one or more tactics (`Tactic`) to achieve its objective. Tactics execute actions (`Action`), which implement the actual adaptation operations to be executed. Actions can be further parameterized by a set of input and output parameters (`Parameter`). Note that the actions do not implement the logic of the operation but trigger the respective adaptation operations when being interpreted. A tactic is specified by an adaptation plan (`AdaptationPlan`), which defines the control flow of executing adaptation actions. In addition to actions to be executed (`ActionReference`), an adaptation plan may contains other common control flow elements (`AbstractControlFlowElement`) such as loops (`Loop`) and branches (`Branch`). We will not further detail the remaining parts of the meta-model, as they are not relevant to this thesis.

Chapter 4

Quality of Service Evaluation and Capacity Management

A software system's quality of service (QoS) denotes the degree to which it satisfies requirements with respect to non-functional runtime characteristics, such as performance, availability, and reliability [Becker et al., 2006b]. QoS evaluation techniques can be divided into approaches based on measurements, simulation, and analytical modeling [Jain, 1991; Lilja, 2005]. Starting with an introduction of basic QoS terminology and measures, this chapter gives an overview of QoS measurement, modeling, and prediction—with a focus on performance and capacity management.

This chapter is organized as follows. Section 4.1 provides an introduction into QoS terminology, characteristics, and measures, as well as the specification of service level agreements. Section 4.2 covers relevant foundations of performance measurement. Section 4.3 presents the capacity planning methodology by Menascé and Almeida [2002], which focuses on offline, long-term scenarios. Sections 4.4 and 4.5 focus on workload characterization and forecasting, as well as performance modeling and prediction.

4.1 Quality of Service

Informally, a system's quality of service (QoS) can be considered the degree to which a system's runtime quality properties satisfy its stakeholders' requirements. Different QoS definitions exist, some of which limit the included categories of properties to non-functional ones [Becker et al., 2006b]; others include functional properties as well [Taylor et al., 2009].

In this thesis, we will focus on selected non-functional QoS measures from a selected set of QoS characteristics, as detailed in the following Section 4.1.2. Section 4.1.3 details the specifications of service level agreements. Section 4.1.4 describes the OMG's Structured Metrics Meta-Model (SMM) and our MAMBA extension and tool support for SMM, which can be used together for representing measures and measurements, and for conducting model-based QoS analysis.

4.1.1 Terminology

A framework for software product quality is provided by the ISO/IEC International Standard 9126 [ISO/IEC, 2001, 2003a,b, 2004]. It proposes a quality model whose main entities are *characteristics*, *subcharacteristics*, and *metrics*. Software quality attributes are categorized into characteristics. Characteristics are further decomposed into subcharacteristics. Subcharacteristics can be measured by metrics. For a basic set of six different characteristics—functionality, reliability, efficiency, etc.—the standard provides basic sets of subcharacteristics—fault tolerance, time behavior, resource utilization, etc.—and metrics (availability and response time, etc.). The standard further distinguishes between *internal quality*, *external quality*, and *quality in use* (metrics). Internal quality considers quality attributes without executing the system, e. g., based on static analysis (complexity expressed in lines of code (LOC), etc.). As opposed to this, external quality and quality in use consider the quality attributes observed from the running system; for quality in use, it is additionally assumed that the system executes in its desired environment.

With respect to terminology in this thesis, we follow the structure of the quality model, i. e., the distinction between characteristics, subcharacteristics, and metrics. However, we favor the notion of *measure* over *metric* because of its narrow meaning in mathematics. In our understanding of a measure, we follow Zuse [1998] who provides a thorough definition of measures and scale types based on empirical and numerical relation systems. This framework is also described and used by Liggesmeyer [2002]. We focus on non-functional external and quality in use attributes under the term quality of service (QoS). Relevant QoS characteristics and measures (independent of the ISO/IEC Std. 9126) are covered in the following Section 4.1.2.

4.1.2 Selected QoS Characteristics and Measures

This section briefly introduces the QoS characteristics *performance*, *capacity*, *efficiency*, *scalability*, and *elasticity*, which are relevant to this thesis. Additionally, we include the term *workload*, which is not a QoS characteristic but one of the main influence factors to many QoS characteristics. We provide and discuss some definitions of the considered characteristics and measures from the literature. For a thorough introduction to QoS characteristics and measures, readers may also refer to one or more of the various textbooks available on this topic (e. g., [Jain, 1991; Smith and Williams, 2002; Menascé and Almeida, 2002; Lilja, 2005]).

Workload

The *workload*, also denoted as the *usage profile* or the *operational profile* [Musa, 1993], refers to the way systems—or one of its components or services—are used. Mainly originating from classic queueing theory [Balsamo and Marin, 2007], some common workload terminology and measures established throughout the past decades. The terms *customer*, *user*, *job*, and *request* are synonyms for a unit of work that arrives at a system for being serviced. Workloads are usually distinguished between *open workloads* and *closed workloads*. For open workloads, it is assumed that independent customers from a *population* enter and depart from the system. In a closed workload, a finite population of customers continuously arrives, departs, and reenters the system. Regardless of whether an open or closed workload is assumed, workload descriptions are often divided into *workload intensity* and *resource demanding* characteristics [Menascé and Almeida, 2002]. The former refers to the characteristics of the number of requests in a certain time period; the latter refers to the properties of the individual requests, e. g., the size of files requested. Important intensity measures for open workloads are the *arrival rate*—the number of customer arrivals per time interval—and its reciprocal value, the *inter-arrival time*. In addition to the population size, closed workloads often include a specification of *think times*, i. e., the time interval between departing and reentering the system.

The global workload a system is exposed to may be comprised of multiple *workload classes* or *workload components* with different characteristics, e. g., open and closed workloads with different workload intensity and

resource demanding characteristics. This set of workload classes and their frequencies of occurrence is typically denoted by the term *workload mix*.

Performance

A wide-spread definition for the term performance is provided by Smith and Williams (Definition 4.1 below). The authors' decision to limit their definition to the subcharacteristic of timeliness is based on the focus of the book that includes this definition; they explicitly mention that other definitions include additional performance subcharacteristics [Smith and Williams, 2002].

Definition 4.1 (Performance [Smith and Williams, 2002])**.** *"Performance is the degree to which a software system or component meets its objectives for timeliness."*

We consider performance to be divided into the two subcharacteristics time behavior and resource usage. This complies to the afore-mentioned ISO/IEC Std. 9126—even though, the standard uses the term *efficiency* instead of *performance*.

Important time behavior measures are *response time* and *throughput*. The response time denotes the time needed by a system to answer a service request. For executions of software operations, the response time is the time interval elapsed between the start and the end of the execution. Throughput denotes the number of service requests processed by a system or a component within a given time interval.

With respect to measures for resource usage, first a basic definition of a *resource* is needed. According to Woodside et al. [2007], a resource is a "system element that offers services required by other system elements." Classes and examples of resources include hardware resources (e. g., CPU, memory, storage), logical software resources (e. g., buffers, semaphores), and processing software resources (e. g., processes, threads) [Woodside et al., 2007]. For resources that are either busy or idle at a point in time (e. g., CPU), the *resource utilization* measure denotes the fraction of time a resource is busy [Jain, 1991] in a given time interval. For resources providing a certain capacity (e. g., memory or storage), the resource utilization measure denotes the fraction of the used and the totally available capacity at a given point in time—which may be further aggregated over a time interval [Jain, 1991]. In addition to these classic measures of resource usage, we also consider

the number of resources of a certain type as a measure of resource usage. This is particularly relevant for virtualized environments, e. g., IaaS cloud infrastructures (Section 3.3.2).

Capacity

Jain [1991] distinguishes three different measures of capacity—*nominal capacity*, *usable capacity*, and *knee capacity*—which are defined based on the relation of workload to the afore-mentioned performance measures throughput and response times. We will extend Jain's definition of the first two measures by including other QoS characteristics in addition to performance. A system's nominal capacity can be defined as the maximum workload intensity a system is able to service. However, a system operating under such workload conditions typically violates QoS requirements (SLAs). Under the constraint that QoS requirements need to be satisfied, the maximum workload intensity is called a system's usable capacity. The knee capacity denotes the workload intensity level at which response times start to increase considerably—even though, they may still satisfy QoS requirements.

Including the monetary cost aspect, Menascé and Almeida [2002] suggest the notion of *adequate capacity* as quoted in Definition 4.2 below. The cost aspect is also relevant for the following QoS characteristic efficiency and for capacity management processes, as detailed in Section 4.3.

Definition 4.2 (Adequate capacity [Menascé and Almeida, 2002])**.** *"A [. . .] system has adequate capacity if the SLAs are continuously met [. . .], and if the services are provided within the cost constraints."*

Efficiency

Efficiency is a QoS characteristics that relates other QoS characteristics to associated costs. An informal definition of efficiency is provided by Taylor et al. [2009] (Definition 4.3).

Definition 4.3 (Efficiency [Taylor et al., 2009])**.** *"Efficiency is a quality that reflects a software system's ability to meet its performance requirements while minimizing its usage of the resources in its computing environment. In other words, efficiency is a measure of a system's resource usage* economy*."*

Note that in this thesis, we use the terms efficiency and resource efficiency as synonyms. Different measures of efficiency exist—typically expressed as a fraction of a QoS measure and a direct or indirect measure of resource usage. Resource usage may be related to the afore-mentioned system resources, e. g., CPU, but also to the use of energy or monetary resources. Equations (4.1) and (4.2) list two—more or less concrete—efficiency measures. Equation (4.1) use a measure of efficiency that is the fraction of an application utilization measure (e. g., request throughput) and power consumption. Note that both values are normalized to a value between zero and one based on peak values. The Space, Watts and Performance (SWaP) measure [Sun Microsystems, 2009] additionally includes the dimension of space usage, e. g., the number of rack units used by the servers.

$$\text{Efficiency} = \frac{\text{Throughput}}{\text{Power Consumption}} \tag{4.1}$$

$$\text{SWaP} = \frac{\text{Performance}}{\text{Space} \times \text{Power Consumption}} \tag{4.2}$$

Scalability

Definition 4.4 below provides a definition for the term scalability by Smith and Williams [2002]. They see scalability as one dimension of timeliness included in Definition 4.1. Hence, for them scalability is a subcharacteristic of performance. Other definitions of scalability are provided by Weinstock and Goodenough [2006].

Definition 4.4 (Scalability [Smith and Williams, 2002])**.** *"Scalability is the ability of a system to continue to meet its response time or throughput objectives as the demand for software functions increases."*

We follow the approach by Duboc et al. [2007] who consider scalability to be a "meta-quality" characteristic of other quality characteristics (Definition 4.5):

Definition 4.5 (Scalability [Duboc et al., 2007])**.** *"We define scalability as a quality of software systems characterized by the causal impact that scaling aspects of the system environment and design have on certain measured system qualities as these aspects are varied over expected operational ranges."*

Elasticity

Elasticity is a QoS characteristic that gained attraction in recent years, particularly in the context of research on self-adaptive approaches for virtualized computing environments, e. g., clouds. Herbst et al. [2013b] propose the following definition for the term elasticity:

Definition 4.6 (Elasticity [Herbst et al., 2013b])**.** *"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible."*

Elasticity is related to the aforementioned QoS characteristics scalability and efficiency. Herbst et al. [2013b] suggest to divide elasticity into *speed* and *precision* aspects, and propose respective measures.

4.1.3 Specification of Service Level Agreements

Particularly in business-critical domains, service providers and consumers often negotiate service level agreements (SLAs), which constitute contractual specifications of QoS requirements and penalties that are due in case these QoS requirements are violated. The QoS requirements as part of the SLAs are also denoted as service level objectives (SLOs). An example for a performance SLO is that *for a defined service, 95 % of all response times observed within any 30 minute time-window must not exceed 500 milliseconds.* In this thesis, we will not focus on penalties and often use the term SLA as synonym for SLO. In addition to the afore-mentioned SLOs and penalties or other actions that are due in case of SLO violations, SLAs typically include additional information, e. g., a definition of the involved parties and their roles, the temporal scope, and references to technical services or interfaces.

For the specification of SLAs, (machine-processable) languages—which can be considered modeling languages in the context of MDSE (Chapter 2)—and corresponding frameworks have been developed in academic and industrial contexts. The core concepts and characteristics of popular approaches are summarized in the remainder of this section (in historical order).

WSLA

Web Service Level Agreement (WSLA) [IBM, 2003; Keller and Ludwig, 2003], initially proposed by IBM in 2001, provides a framework for defining and monitoring SLAs. A WSLA document is structured into three parts: definitions of parties, services, and obligations. In addition to service providers and consumers, the specification explicitly allows the involvement of third parties, e. g., providing measurement or condition evaluation services. In the service definition part, SLA parameters specify measures for services, how measurements are obtained from managed entities, or how composite measures are computed from other measures. These SLA parameters are used in the obligations part of the document, in form of service level obligations and action guarantees. The latter specify constraints on how and when an obliged party must invoke a party's action interface also defined in the WSLA document. The WSLA language is defined as an XML schema, allowing domain-specific extensions. It already includes a number of standard extensions for web services, e. g., WSDL/SOAP action descriptions, measurement directives (counter, response time, invocation count, etc.), and functions (time series constructors, mean, median, mode, etc.).

WSOL

Web Service Offerings Language (WSOL) [Tosic et al., 2002; Tosic, 2004] is a language that allows the formal specification of constraints for web services. For a web service described using the Web Services Description Language (WSDL) [World Wide Web Consortium (W3C), 2007b], WSOL allows to define different service classes that may, for example, differ in pricing, usage privileges, or guaranteed QoS levels. WSOL supports the definition of *a*) functional, *b*) QoS, and *c*) access right constraints. In order to use WSOL for specifying SLAs, included (periodic) QoS measures can be used in constraint expressions. Management statements can be used to express pricing information for service usage or penalties due when constraints are violated. Like WSDL, WSOL is implemented as an XML schema. The Web Service Offerings Infrastructure (WSOI) [Tosic, 2004] provides a tooling infrastructure for WSOL, supporting the measurement and calculation of QoS measures, evaluation of constraints, etc.

WS-Agreement

Web Service Agreement (WS-Agreement) [Open Grid Forum, 2011] is a standard published by the Open Grid Forum (OGF). An initial version was published in 2003. It depends on other *WS-** web service specifications, e. g., WS-Addressing. WS-Agreement allows to formulate SLA documents including usual contents like temporal scopes of the agreement, involved parties, service definitions/references, as well as the expression of guarantees and terms for the services in form of promises and penalties. The standard includes a protocol for negotiating SLAs: a so-called *agreement factory* offers the structure of agreements to accept in form of WS-Agreement templates, which clients use to make concrete offers/requests. Also, the standard proposes runtime states for agreements and terms that can be used for monitoring the compliance of SLAs. WS-Agreement is specified as an XML schema, allowing expression languages to be used for formulating constraints etc.

SLA*

SLA* [Kearney et al., 2010] was developed as part of the FP7 ICT project *SLA@SOI*. It supports the definition of involved parties, interface declarations of offered services, and QoS guarantees for these services—including actions to be taken by respective parties, e. g., payments due if QoS guarantees are violated. SLA* includes an expression language for an extensible vocabulary of nominals (e. g., protocols, data types, and units), functions (e. g., arithmetic/set operations, time series, and QoS measures), and classes of events (e. g., violations and warnings). Example QoS measures already included are availability, arrival rate, throughput, completion time, mean time to failure (MTTF), and mean time to repair (MTTR). Example set operations are sum, standard deviation, median, and mode. Kearney et al. [2010] provide a set-based specification of the SLA*'s abstract syntax, for which a Java API, an XML schema, and a grammar in Backus–Naur Form (BNF) exist. The implementation is online.[1] An example SLA* (template) specification is shown in Figure 4.1.

[1] `http://sourceforge.net/apps/trac/sla-at-soi/`

```
01: slatemplate{
02:   version : sla-star-v1
03:   vocabularies :
04:     http://sla-at-soi.eu/core
05:   parties :
06:     Fred : party{
07:       role : provider
08:     }
09:   interfaceDeclrs :
10:     IF1 : interfaceDeclr{
11:       provider : Fred
12:       endpoints :
13:         E1 : endpoint{
14:           location : fred@xyz.com
15:           protocol : email
16:         }
17:       interface : http://xyz.com/service
18:     }
19:   variables :
20:     S : var{
21:       expr : IF1/request
22:     },
23:     X : var{
24:       expr : basic
25:       domain : one-of { premium, basic }
26:     }
27:   terms :
28:     AT1 : agreementTerm{
29:       pre : arrival-rate(S) <= 2 tx_per_day
30:       G1 : state{
31:         pre : X == basic
32:         post : completion-time(S) < 1 hr
33:       },
34:       G2 : state{
35:         pre : X == premium
36:         post : completion-time(S) < 10 min
37:       },
38:       G3 : action{
39:         actor : Fred
40:         policy : mandatory
41:         pre : violated[ G1.post and G2.post ]
42:         limit : 1 week
43:         post : payment{
44:           recipient : customer
45:           value : 1 euro
46:         }
47:       }
48:     }
49: }
```

Figure 4.1. Example SLA* (template) specification [Kearney et al., 2010] (BNF serialization)

SLAng

SLAng [Skene, 2007; Skene et al., 2010] focuses on the SLA specification in application service provisioning scenarios. It supports the definition of parties, services, penalties, and clauses involved in SLAs. The implementation of the language is based on the OMG's EMOF, OCL, and HUTN specifications, mentioned in Section 2.3.1. The language is extensible in order to support domain-specific additions. An initial version of the SLAng language was presented in 2004 [Skene et al., 2004]. Since then, it evolved considerably, as described by Skene [2007]. The language specification and supporting tools are available on the SLAng web site [Skene, 2014].

4.1.4 SMM and MAMBA

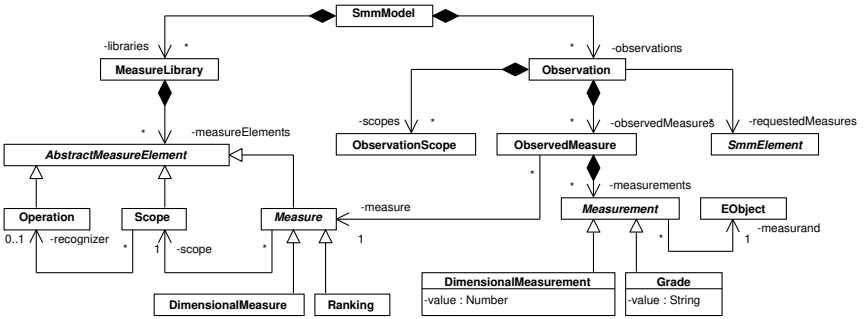
The Structured Metrics Meta-Model (SMM) [Object Management Group, Inc., 2012b] is one of the meta-model specifications developed by the Architecture-Driven Modernization (ADM) Task Force [Object Management Group, Inc., 2013c], a sub-committee of the OMG (see also Section 2.3.1). SMM provides generic means to specify elements relevant for the domain of model-based measurement. In the context of this thesis, we extended SMM

by additional meta-model constructs and extensible tool support, summarized under the approach called MAMBA. The remainder of this section first provides a brief overview about the SMM specification, describes our identified shortcomings that were the motivation for the work on MAMBA, which is described thereafter. Note that this section is largely based on our publications on MAMBA [Frey et al., 2011, 2012] and contains parts of these publications.

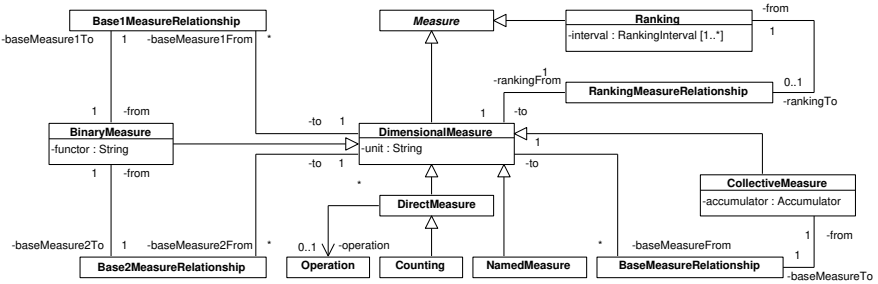
SMM Overview

The SMM specification employs the central notion of *measures* to describe methods for computing values upon MOF-based models. According to Section 4.1.1, the term is used as a synonym for *metric*, in the sense that it describes an algorithm for calculating specific properties of a software system's elements. A software function's cyclomatic complexity, the contained lines of code, or its average response time constitute classic examples of such measures. The meta-model defined by SMM enables to specify those measures—along with the computed results and further concepts specific for the measurement domain—following an abstract representation that facilitates its utilization as a common interchange format that can be used by different tool vendors. Referring to Figure 4.2, which includes the meta-model's core classes and relationships, we will provide a brief overall overview of SMM. For details, please refer to the specification [Object Management Group, Inc., 2012b].

A `Measure` can be applied to a specific set of model elements that is defined through the `Scope` class. For example, a `Scope` may limit the computation of the cyclomatic complexity to source code snippets and therefore exclude BLOB files that can be present in extracted system models. In addition, to consider only a subset of the source files (e. g., those containing C source code), an `Operation` could be formulated that specifies this restriction using OCL. The classes of SMM mentioned so far inherit from the `AbstractMeasureElement` class. Corresponding child classes can get registered and be supplied via a `MeasureLibrary`. Furthermore, Figure 4.2a illustrates the two concrete measures `Ranking` and `DimensionalMeasure`. The first may, for example, classify a method as *low prio* or *high prio* depending on its cyclomatic complexity residing in $[0, 40)$ or $[40, \infty)$, respectively. The `DimensionalMeasure` describes a measure that assigns a numeric value. The counterpart of a



(a) Core elements and relationships in the SMM meta-model



(b) SMM's `Measure` classes

Figure 4.2. Core SMM meta-model parts, including `Measure` classes [Frey et al., 2011]. Deviating from the SMM specification, `EObject` instead of `MofElement` is used as a `measurand`'s type.

`Measure` is a corresponding `Measurement` that holds the result being produced through executing the `Measure`. The counterparts of the previously mentioned classes `Ranking` and `DimensionalMeasure` are the `Measurement`s `Grade` and `DimensionalMeasurement`. A model element that was measured is referenced via the `measurand` relationship of the respective `Measurement`. This could be a specific method model element, for instance. As we want to measure Ecore-based models (Section 2.3.2), `measurand` points to an `EObject` in the context of MAMBA. A concrete measurement process is encapsulated via an instance of `Observation` and can therefore be distinguished from other measurement

runs. The *Observation* contains information such as the time of the measurement and it describes the actually used *Measures* and the measured elements through referencing *ObservedMeasure* and *ObservationScope*, respectively.

Further measures that are relevant for our SMM extensions are presented in Figure 4.2b. A *DirectMeasure* can measure a model element through applying an *Operation*. *Counting* is a specific *DirectMeasure*. It is used to restrict its *Scope* to a relevant subset of model elements as the referenced *Operation* returns 0 or 1 for a given *measurand*. A *NamedMeasure* denotes a familiar measure that can be described unambiguously by solely stating its name. *BinaryMeasure*s apply two measures—referenced via *Base1MeasureRelationship* and *Base2MeasureRelationship*—to a model element and then evaluate a binary function upon the corresponding measurements, for example, to calculate their difference. A *CollectiveMeasure* enables to apply an accumulator to any number of collected base measurements, e. g., to compute the standard deviation.

SMM Shortcomings

The SMM specification contains the *CollectiveMeasure* class modeling the accumulation of measurements for an associated base measure (*DimensionalMeasure*) into a single value. However, using the *Accumulator* enumeration, the SMM specification limits the set of supported aggregate functions to *sum*, *maximum*, *minimum*, *average*, and *standard deviation*. This way, common aggregate functions, such as median or other percentile functions, cannot be used with SMM so far.

Moreover, especially in service-level management, collective measures are applied to bounded sets of contiguous base measurements. For example, in SLA documents the definition of QoS measures, e. g., addressing availability and performance, is based on time periods. The main reason is that short-term QoS degradations are hidden by aggregations over long-term periods. The corresponding aggregate function is then applied in periodic time steps, considering base measurements observed during the elapsed time period of specified length. Currently, the SMM specification has no support for modeling periodic measures of any kind. A further challenge when adopting SMM's current version 1.0 can be seen in the lack of the specification's maturity. Although the basic structure and ideas of SMM are encouraging, there still exist some inconsistencies that currently impede its

interoperability and that should be addressed.

With our MAMBA approach, we aim to address some of the SMM's current shortcomings. Table 4.1 provides a compact comparison of pure SMM and MAMBA.

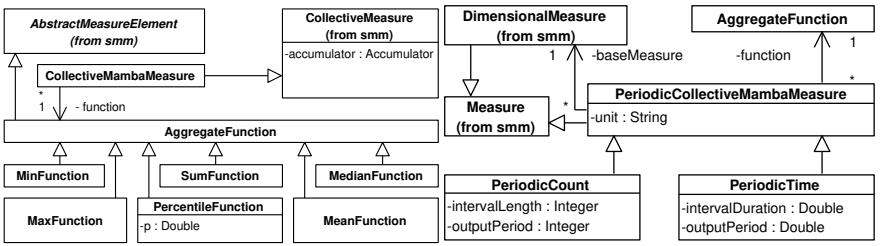
Table 4.1. Comparison of pure SMM and MAMBA [Frey et al., 2011]

Issue	**Pure SMM**	**MAMBA**
Meta-models	MOF	Ecore
Model execution	Limited tool support	MEE
Raw measurement data integration	No tool support	Via measurement providers
Model querying using measurements	No native support	MQL: implicit calculation and integration of measurements in queries
Periodic measures	No native support	Periodic collective MAMBA measures
SMM as runtime model	No native support	Continuous execution with MEE
Aggregate functions	*Limited set*: sum, maximum, minimum, average, standard deviation	*Extensible set*: sum, maximum, minimum, average, standard deviation, median, percentile, etc.

MAMBA Meta-Model Extensions to SMM

In order to support *a*) arbitrary aggregate functions, *b*) extended collective measures, and *c*) periodic collective measure, MAMBA includes the below-described extensions to the SMM meta-model, which are also depicted in Figure 4.3.

- ▻ *Arbitrary Aggregate Functions.* In order to make the set of supported aggregate functions extensible, we added a new meta-model class `AggregateFunction` (abstract) into our SMM extension, as shown in Figure 4.3a. Also included are six example aggregate functions. The parameterized `PercentileFunction` class demonstrates the limitation of using enumerations or strings to select aggregate functions.



(a) Aggregate functions and collective measures **(b)** Periodic collective measures

Figure 4.3. MAMBA extension mechanism for aggregate functions as well as collective and periodic measures [Frey et al., 2011]

MAMBA users can now use these or custom aggregate functions in (periodic) collective measures, as detailed in the following paragraphs. Custom functions can be defined by using meta-model classes that extend `AggregateFunction`.

- *Extended Collective Measures.* Since SMM's `CollectiveMeasure`s cannot use the newly introduced `AggregateFunction`s, we added another meta-model class `CollectiveMambaMeasure`. `CollectiveMambaMeasure` extends the SMM class `CollectiveMeasure` and references an `AggregateFunction`, introduced above. Figure 4.3a shows the meta-model extensions regarding the newly introduced collective measure and associated aggregate function.
- *Periodic Collective Measures.* We included support for modeling periodic measures in our SMM extension by introducing the abstract meta-model class `PeriodicCollectiveMambaMeasure`. Just like the `CollectiveMambaMeasure` class described above, it references a MAMBA aggregate function (`AggregateFunction`). This meta-model extension is depicted in Figure 4.3b. `PeriodicCollectiveMambaMeasure` extends `Measure` rather than `DimensionalMeasure` (or `CollectiveMambaMeasure`) in order to preclude semantic ambiguity, for instance because the latter would allow `PeriodicCollectiveMambaMeasure` to be used as base measure of collective measures. Currently, two concrete classes for periodic collective measures are included in our SMM meta-model extension (see also Figure 4.3b): `PeriodicTime` and `PeriodicCount`. `PeriodicTime` can be used to model measures where the

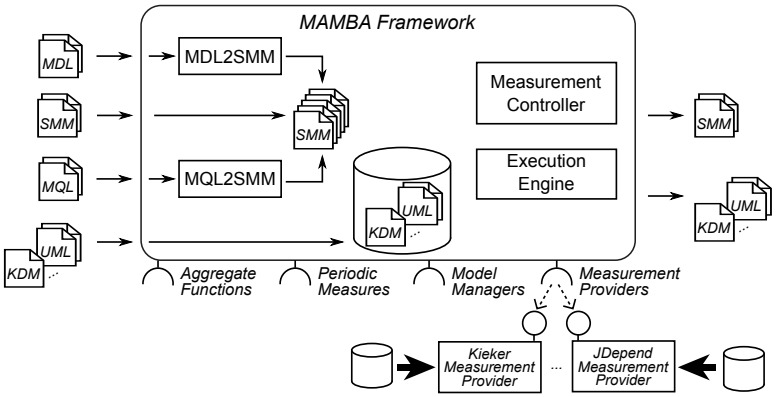


Figure 4.4. MAMBA framework with measurement providers

referenced aggregate function is repetitively applied with a time period (*outputPeriod*) incorporating the measurements observed within the elapsed time period of length *intervalDuration*. `PeriodicCount` triggers the aggregate function to be computed for every *outputPeriod*-th new measurements, incorporating the past *intervalLength* measurements.

Tool Support—MAMBA Framework

Figure 4.4 provides an architectural view of the MAMBA framework in terms of core components and usage. The general idea is that users provide a set of domain models, e. g., instances of the Knowledge Discovery Meta-Model, along with a definition of requested measures in form of SMM models or textual representations which MAMBA translates to SMM. The framework executes the input SMM models and outputs these SMM models enriched by measurements (observations) for the requested measures. Note that the SMM models may contain additional MAMBA-provided extensions, such as the aforementioned user-defined (periodic) aggregate functions. Measurement Providers are used to integrate external static or dynamic analysis tools, e. g., by executing these and importing the resulting raw measurement data. Figure 4.4 indicates the integration of the dynamic

and respectively static analysis tools Kieker (Chapter 7) and JDepend.[2]

The Measurement Controller creates an instance of the Execution Engine and passes the SMM models, including the list of requested measures, to the latter. As a first step, the Execution Engine inspects the SMM models in order to determine whether at least one named measure is required for the computation of the requested measures. We distinguish between two different modes of execution: if no dependency to a named measure exists (*closed mode*), the Execution Engine can directly compute the SMM measurements simply based on the domain model(s); otherwise (*open mode*), the Execution Engine provides the list of required named measures to the Measurement Controller, which initializes appropriate Measurement Providers for these named measures. The Measurement Providers create observations (i. e., measurements for named measures, including additional meta-information) that are passed to the Execution Engine via the Measurement Controller. After the termination of each Measurement Provider, the Execution Engine executes the SMM models just like the way it executes in closed mode.

4.2 Performance Measurement

Measurement-based performance evaluation techniques obtain values for performance measures of interest—e. g., response times, throughput, and resource utilization (Section 4.1.2)—by collecting, processing, and analyzing runtime data from a system under execution. This chapter provides a brief introduction into selected aspects of performance measurement with respect to trigger mechanisms (Section 4.2.1), monitors and instrumentation (Section 4.2.2), perturbation (Section 4.2.3), and monitoring enterprise application systems (Section 4.2.4). For a detailed presentation of foundations on performance measurement of software systems, we suggest to refer to Jain [1991], Lilja [2005], as well as Menascé and Almeida [2002]. Basic principles of measurement—not limited to the domain of software systems—is provided by the Joint Committee for Guides in Metrology (JCGM) [2008]. The contents of this section are mainly based on these sources. Readers may also refer to the aforementioned source for definitions of basic measurement terminology, such as accuracy, precision, resolution, etc.

[2] http://clarkware.com/software/JDepend.html

4.2.1 Trigger Mechanisms

Mechanisms to trigger measurements of relevant data from a system can be divided into *event-driven* and *sampling-based* strategies [Lilja, 2005; Menascé and Almeida, 2002]. Event-driven techniques collect or update measurements whenever a relevant event in the system occurs. Example events include invocations of software operations or the occurrence of exceptions. In the simplest case, an event-driven measurement routine updates an event counter. A special event-driven technique is *tracing*, which involves the collection of data about an event and the respective current system state. For example, start and end times, as well as a transaction identifier may be recorded for executed software operations. Sampling-based measurements are not triggered by the occurrence of system events but they are executed at fixed time intervals. Hence, only snapshots of system states are taken, which provides less detailed information compared to event-driven techniques. Both event-driven and sampling-based techniques may either provide the measures of interest directly or they have to be computed indirectly by other measures.

4.2.2 Monitors and Instrumentation

Tools used to collect measurement data of interest from a software system are referred to as *monitors* [Menascé and Almeida, 2002]. According to Jain [1991], monitors can be classified based on the trigger mechanism, the result display ability, and the implementation level. Trigger mechanisms have been discussed in Section 4.2.1 already. The result display ability defines whether the collected data is displayed/processed online or offline. With respect to the implementation level, monitors can be roughly divided into *hardware monitors* and *software monitors* (hybrid solutions exist as well). As indicated by the name, hardware monitors are implemented in hardware and focus on low-level measurements based on electrical signals and hardware registers. They are typically part of hardware devices, e. g., CPUs, memory, and hard disk drives. As opposed to this, software monitors are software routines integrated in the analyzed software system. The process of integrating software monitors into a system is called *instrumentation*. Instrumentation may be added to the application (source/object/byte) code or the underlying runtime environment in form of operating system, middleware, or application server.

Various instrumentation techniques exist, e. g., direct code modification, indirect code modification using compiler modification or aspect-oriented programming (AOP), or middleware interception [Jain, 1991; Lilja, 2005; Kiczales et al., 1996; Menascé and Almeida, 2002]. Often, software-based logging mechanisms serving useful performance measurements are already built in to the runtime environment, e. g., access logs of web servers.

Techniques to add instrumentation to Java applications include low-level byte code manipulation, e. g., based on ASM[3] and BCEL,[4] AOP-based libraries such as AspectJ [Kiczales et al., 2001], or higher level instrumentation languages such as DiSL [Marek et al., 2012]. The JVM includes the JVM Tool Interface (JVMTI) [Oracle, 2011], which provides an API for controlling and monitoring the state of Java applications. Many debuggers and profilers make use of this interface. Various measures about the JVM and executing Java applications can be accessed via JMX (see also Section 3.3).

4.2.3 Perturbation

Perturbation is an important aspect to be considered when planning and executing performance measurements in software systems. This is due to the fact that the instrumentation and the execution of monitors may alter the system's runtime behavior. This is particularly true for software monitors, which compete for shared resources with the system under analysis, e. g., CPU, memory, and storage, or change the software control flow due to measurement routines. Perturbation that has an impact on a system's performance properties, i. e., timing behavior and resource usage, is often referred to as *overhead*. However, monitors may also have an impact on other QoS characteristics, e. g., reliability due to implementation errors.

The degree of perturbation introduced by measurements depends on different aspects, e. g., the measurement strategy—event-driven vs. sampling-based—, the granularity of instrumentation, as well as the types and quality of monitors used. Particularly, software-based tracing can introduce considerable perturbation because it may comprise the collection and storage/-transfer of large amounts of data, imposing demands to I/O and processing resources. On the other hand, it is sufficient to maintain simple counters for selected event-driven and sampling-based measurements.

[3] `http://asm.ow2.org/`

[4] `http://commons.apache.org/proper/commons-bcel/`

The acceptable degree of perturbation differs depending on the performed measurement-based application. Typical measurement-based applications for software system development and operation are debugging, profiling, logging, and monitoring. Debugging and profiling are usually performed at development time in development environments where a high degree of perturbation is acceptable. For example, profiler tools typically impose a high performance overhead but provide very detailed information on the runtime behavior. Also, experimental tools may be used to gain the desired information. On the other hand, logging and monitoring are used during operation in the production environment, which limits the accepted perturbation to a level that does not violate the system's SLAs. Administrators usually accept a certain level of perturbation because they gain important information about the runtime behavior and system health.

4.2.4 Monitoring of EASs

Continuous monitoring of enterprise application systems (EASs) serves, for example, to make sure that the system's QoS requirements are fulfilled as well as to detect, diagnose, and resolve QoS problems as early as possible. For this purpose, monitors are placed at different layers of the software system stack, including network devices, server and storage hardware, the operating system, middleware and application servers, applications, as well as business processes. On each level, various QoS measures of interest exist. While these are typically EAS-agnostic for system level measurements, e. g., utilization of CPUs and storage resources, the set of measures becomes EAS-specific, when it comes to application or business process level, e. g., involving measures like completed orders per hour.

Countless monitoring tools have been developed and are in production use since the past decade at least. For example, Allspaw [2008] provides a set of best practices, including suggestions of tools, for monitoring EASs. Various powerful commercial tools are offered under the umbrella of the term application performance management (APM) [Kowall and Cappelli, 2013].

4.3 Capacity Management

This section gives an overview of typical capacity management activities and techniques. We use the terms capacity planning (e. g., used by Menascé and Almeida [2002]), capacity management (e. g., used in the ISO/IEC Standard 20000 [ISO/IEC, 2005b,a]), and resource management (e. g., used by Kounev et al. [2010]) as synonyms.

Figure 4.5 depicts the capacity planning methodology by Menascé and Almeida [2002]. It includes three models—representing information on workload, performance, and costs—as well as respective steps to create, calibrate, and use these models for prediction. Based on the focus of this section, we limit the QoS characteristic to be analyzed as part of this capacity methodology to performance. Menascé and Almeida additionally include availability. Based on our understanding of capacity (e. g., Definition 4.2), it makes sense, to extend the methodology to other QoS characteristics, based on appropriate models and analysis techniques.

The initial step in the process, understanding the environment, comprises activities such as identifying the technical hardware and software infrastructure, integration with other systems, and the SLAs.

The workload characterization step serves to create a workload model for the system, which represents the usage profile of the system. This includes

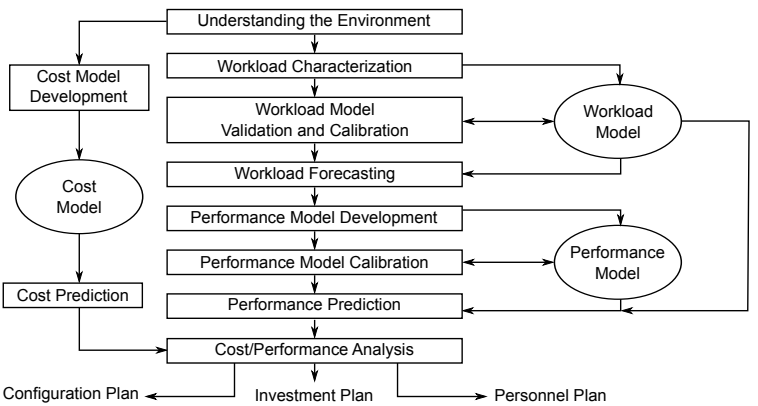


Figure 4.5. Capacity planning methodology by Menascé and Almeida [2002]

the identification of the workload classes (open, closed) as well as their characteristics with respect to workload intensity and resource demands for services provided by the system. The workload characterization process typically comprises a combination of measurements (Section 4.2) and expert knowledge. The goal of the workload model validation and calibration step is to assess the validity of the workload model w.r.t. the desired level of accuracy and, if needed, to refine the model to reach this accuracy level (calibration). Having observed a system's workload measures for a certain period of time allows to predict future workload measures using the workload model. This step is called workload forecasting. We will present selected techniques for workload modeling and forecasting in Section 4.4.

A performance model is an abstract representation of a system's performance-relevant characteristics. For example, the system may be modeled in terms of interactions among software entities and their demands to shared hardware resources such as CPU, network, and I/O. As for the workload model, the capacity planning process includes development, calibration, and prediction. Predictive values for performance measures of interest can be obtained by solving the performance model including information about the usage profile, included in the workload model. We will present selected techniques for performance modeling and prediction in Section 4.5.

The cost model comprises the expenditures for buying, installing, and operating the system in the considered configuration and environment. This includes startup costs, eg, for hardware devices, software licenses and development, as well as operating costs, e. g., for system maintenance, third-party services, and energy. With respect to the corresponding steps in the capacity planning methodology, a costs models is developed and then used for prediction. We will not further detail the topic of cost modeling.

In order to assess whether a system configuration provides adequate capacity, the performance and costs measures obtained from the predictions are compared with the respective constraints and requirements, e. g., as contained in the SLAs.

4.4 Workload Characterization and Forecasting

This section briefly surveys selected workload modeling formalisms and presents selected workload characterization and forecasting techniques.

Note that basic workload terminology and measures have already been introduced in Section 4.1.2.

4.4.1 Workload Modeling

The presentation of the workload modeling formalisms in this section is based on a classification into approaches *a*) for queuing models and variants, *b*) based on scenarios, and *c*) based on sessions. References to performance modeling languages are included, which are introduced in Section 4.5.

▻ *Workload Intensity in Queueing Models and Variants.*

The Kendall notation [Jain, 1991] for Queueing Models (QMs) includes a definition of the workload intensity by specifying the inter-arrival time distribution as well as the population size. Open Queueing Networks (QNs) typically include entry nodes, for which the stochastic arrival processes are defined [Smith and Williams, 2002]. The workload intensity for closed QN can be defined based on population sizes and delay nodes modeling think times. In multi-class QNs, these parameters can be defined for each workload class [Bertoli et al., 2009]. In Layered Queueing Network (LQN) models [Franks et al., 2009], sources of arrivals are modeled as usual tasks (software servers) that request services from other tasks in the LQN model. Open and closed workloads can be modeled by choosing appropriate multiplicities of the tasks and the associated processor nodes.

▻ *Scenario-Based Workload Specification.*

Many architecture-level performance modeling approaches include concepts to define the workload intensity by attaching this information to the modeled performance scenarios. This allows to model frequencies and patterns of occurrence based on the individual performance-relevant use cases. In the SPE modeling approach by Smith and Williams [2002], the performance scenarios are part of the software execution model and arrival rates can be assigned to each of the scenarios. The workload intensity for the system execution model, i. e., QN models, can be derived from the software execution model, e. g., automatically by the corresponding SPE•ED tool. Likewise, the UML SPT profile [Object Management Group, Inc., 2005] allows to associate scenarios with open and closed workloads,

which are parameterized by specifications of arrivals (*occurrence pattern*), *population*, and think times (*external delay*) respectively. An according workload stereotype is assigned to the first step of a scenario. Allowed attribute values include the definition of probability distributions and different arrival patterns, e. g., *bursty*, *bounded*, and *periodic*. In the Core Scenario Model (CSM) [Petriu and Woodside, 2007], an open or closed workload is also defined based on scenarios by associating it with the start step of a scenario. Employing the SPT successor, MARTE [Object Management Group, Inc., 2011c], workloads can be associated to *contexts* by defining streams of events triggering the execution of scenarios. These streams can be specified as timed events, arrival patterns (as for SPT, including closed and open patterns), workload generator models (e. g., UML state machine models), as well as event traces stored in a file.

▻ *Session-Based Workload Specification.*

For certain kinds of systems, the assumption of workload being an arrival of independent requests is inappropriate. For example, many web-based systems employ the concept of *sessions* for users interacting with the systems [Menascé et al., 1999; Goševa-Popstojanova et al., 2006]. Such a session comprises the sequence of inter-related requests posed by a single user during a single visit.

Menascé et al. [1999] introduced the so-called Customer Behavior Model Graph (CBMG) in order to model the navigational patterns of similar sessions. A CBMG consists of a discrete-time Markov chain (DTMC) modeling the transitions between client requests, as well as average think times assigned to these transitions. A set of CBMGs with their relative frequencies of occurrence constitutes the workload mix. In our previous work [van Hoorn et al., 2008], we used these models in combination with an application-specific model of valid sessions for workload generation. In the Palladio Component Model, open and closed workloads are defined *scenario behaviors*, modeling the navigational pattern between parameterized *system calls*. More sophisticated workloads models can be implemented employing MARTE [Object Management Group, Inc., 2011c] by defining appropriate workload generator models, e. g., by means of UML state machines.

4.4.2 Workload Characterization

It is important to keep in mind, that there is neither a general workload characterization methodology nor a general set of workload meta-models or metrics. Instead, these particularly depend on the goal and the abstraction level of the workload characterization process. For example, from a business perspective one might be interested in the number of purchase orders sent per hour, while on a protocol or hardware resource level, characteristics of HTTP requests or I/O operations respectively may be more relevant.

▻ *Characteristics of Web Workloads.*

Some characteristics that are specific for workload intensities and resource demands occurring for systems in the World Wide Web have been identified by researches. This includes burstiness of arrival processes [Arlitt et al., 2001], self-similarity and seasonal patterns in the request rates [Crovella and Bestavros, 1997; Arlitt et al., 2001], heavy-tailed distributions of file sizes and user think times [Crovella and Bestavros, 1997], the difficulty to distinguish real users from robots [Arlitt et al., 2001], as well as the importance of session-based workload characterization [Menascé et al., 1999; Arlitt et al., 2001; Goševa-Popstojanova et al., 2006]

Menascé and Almeida [2002] describe a possible workload characterization methodology for such systems. Possible views on workload can range from business to technology perspectives, implying the workload models to cover business, functionality, protocol, or resource-oriented parameters. Depending on the selected subsystems to be studied (e. g., interactive portals or back-end services), the basic workload components (e. g., user sessions or transactions) as well as the concrete parameters need to be chosen. For the subsequent data collection step, monitoring tools (system and application level) as well as server access logs are typical sources of information. The global workload contained in this data should then be partitioned into workload classes, for example, by grouping requests based on types (images, services, etc.), service names, or resource demands (e. g., estimated by response times).

▻ *Approaches Applying Clustering.*

In order to identify similar workload classes within the global workload mix, clustering is a technique often used. A number of clustering algorithms exists which have been surveyed, for example, by Berkhin [2002]. Menascé et al. [1999] applied *k*-means clustering to identify similar sessions contained in web server access logs. In a first step, the sequence of requests contained in the access log is transformed into a session log—containing a CBMG for each session. The distance metric for the clustering algorithm is defined based on the CBMG's transitions and think time matrices. Arlitt et al. [2001] applied clustering to identify and analyze user sessions by their resource demands—in terms of the number of cacheable, non-cacheable, and search requests.

4.4.3 Workload Characterization and Forecasting Based on Time Series Analysis

Workload forecasting techniques can be divided into quantitative and qualitative techniques [Menascé and Almeida, 2002]. Quantitative approaches use collected values of workload measures to predict future trends, while qualitative approaches are relying on estimates provided by experts. Common approaches for quantitative workload forecasting employ established techniques from time series analysis, which are being used for forecasting in many domains.

A time series X represents a sequence of values $\{x_0, x_1, \ldots\}$ associated with a corresponding sequence of equidistant time points $t = t_0, t_1, \ldots$. For example, the number of requests to system-provided services observed during time intervals of five minutes may be represented as a time series. Established techniques exist to analyze properties of time series, e. g., covariance, correlation, stationarity, linearity, seasonality, as well as to transform time series, e. g., using regression (smoothing, filtering, etc.) and differencing operations. These techniques are useful for workload characterization and forecasting. Recently, a self-adaptive approach for workload characterization and forecasting based on time series analysis has been presented by Herbst et al. [2013a]. Their work includes a classification of forecasting algorithms, including autoregressive integrated moving average (ARIMA) models, for workload forecasting. In the context of this thesis and in collab-

oration with Herbst (Section 5.3), Bielefeld [2012] developed an approach for forecasting response times based on time series analysis in order to detect performance anomalies. For a thorough introduction into time series analysis, readers may refer to one of the many textbooks available on this topic, e. g., by Shumway and Stoffer [2006].

4.5 Performance Modeling and Prediction

This section starts with an overview of approaches for software performance modeling and prediction (Section 4.5.1) before introducing the Palladio Component Model in more detail (Section 4.5.2).

4.5.1 Overview of Approaches

Performance modeling languages and prediction techniques can be grouped into *a*) analytic (or analysis-oriented) and *b*) architecture-level (or design-oriented) approaches. This sections aims to provide a rough overview of the field. For a more detailed introduction into the topic of model-based and model-driven software performance engineering, readers may, e. g., refer to Balsamo et al. [2004], Woodside et al. [2007], and Cortellessa et al. [2011]. Surveys focusing on component-based systems are provided by Becker et al. [2006a] and Koziolek [2010]. References to more specific sources are included below.

Analytic performance modeling languages, many of which are known for decades, typically support very abstract constructs—such as processes, resources, and queues—which are suitable for computer system performance evaluation [Jain, 1991] but have no directly corresponding elements in the software architecture. Efficient techniques and tools to solve such models exist, e. g., based on product-form solutions (approximate or exact) and simulation. Well-known approaches in this category include those that employ variants of Markov processes, Queueing Models (QMs) and Queueing Networks (QNs) [Menascé et al., 2004; Balsamo and Marin, 2007], Petri Nets (PNs) [Bause and Kritzinger, 2002; Balbo, 2007], process algebras [Clark et al., 2007]. Building on this work, approaches such as Layered Queueing Networks (LQNs) [Rolia and Sevcik, 1995; Franks et al., 2009] and Queueing Petri Nets (QPNs) [Bause and Buchholz, 1998; Kounev et al., 2012]

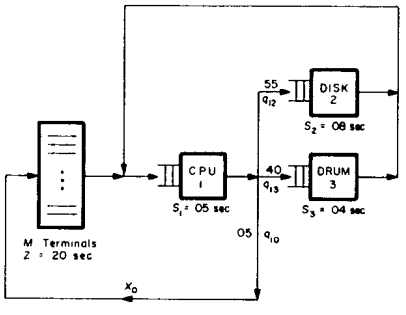
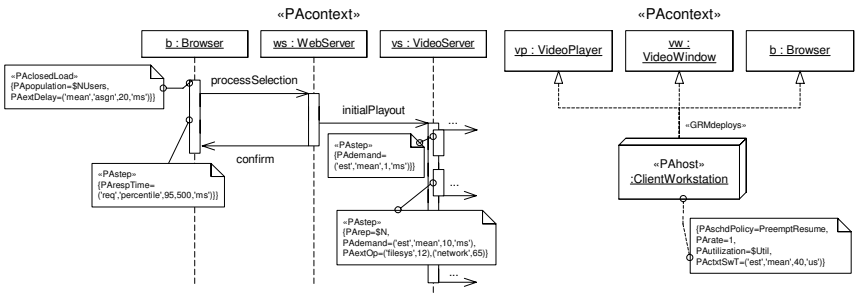


Figure 4.6. Closed Queueing Network [Denning and Buzen, 1978]



(a) Sequence diagram **(b)** Deployment diagram

Figure 4.7. UML SPT sequence (a) and deployment (b) diagrams [Object Management Group, Inc., 2005]

have been developed that aim to improve modeling of software aspects, e. g., software resources. Introductions into a number of tools for these approaches are provided by ACM SIGMETRICS [2009]. As an example, Figure 4.6 shows a QN from the original publication by Denning and Buzen [1978], which models the performance of a computer system with three resources and a closed workload.

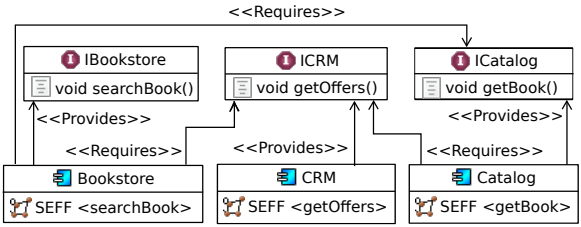
During the past decade, model-based approaches aiming to predict the performance of software systems in early design stages have been developed.

Many of these approaches share the concept of augmenting architectural software design models by performance-relevant properties [Woodside et al., 2002], e. g., resource demands. Examples are the pioneering work by Smith and Williams [2002], OMG's UML SPT [Object Management Group, Inc., 2005] and MARTE profiles [Object Management Group, Inc., 2011c], as well as the CB-SPE [Bertolino and Mirandola, 2004] and the Palladio [Becker et al., 2009] approaches both tailored for the performance prediction of component-based software systems. As an example, Figure 4.7 shows two UML diagrams with SPT annotations, as included in the SPT specification [Object Management Group, Inc., 2005].

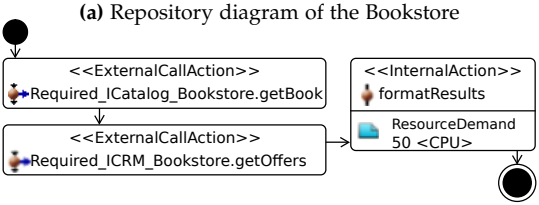
Various transformations from architecture-level performance modeling languages to analytic performance languages have been developed [Di Marco and Mirandola, 2006] to derive quantitative performance indicators for software architectures. Performance interchange formats, e. g., S-PMIF [Smith et al., 2005], KLAPER [Grassi et al., 2007], and CSM [Petriu and Woodside, 2007], aim to bridge the gap between analysis and architecture-level models.

4.5.2 Palladio Component Model

The Palladio Component Model (PCM) [Becker et al., 2009] is a modeling language for architecture-based performance prediction of component-based software systems (CBSSs). A PCM instance consists of four complementary models providing architectural views to structural as well as performance-relevant behavioral aspects of a CBSS: *a*) a *repository* model, *b*) a *system* model, *c*) a *resource environment* model, and *d*) an *allocation* model. Additionally, *usage* models allow to specify corresponding workloads. Transformations from PCM instances to analytic performance models and simulation models exist, allowing to derive performance indices of interest—e. g., statistical distributions of operation response times and resource utilization. The remainder of this section describes PCM's modeling concepts and related terminology required to understand the remaining parts of this thesis. For further details, we refer to the publications on PCM [Becker et al., 2009; Reussner et al., 2011]. Note that this section includes contents from one of our publications [von Massow et al., 2011].



(a) Repository diagram of the Bookstore



(b) RDSEFF of the *searchBook* service

Figure 4.8. PCM repository contents of the Bookstore example application

Repository

A PCM repository model contains the type-level specification of available interfaces and components. An interface constitutes a named set of service signatures, as known from object-oriented modeling. Components *provide* or *require* these interfaces. Figure 4.8a illustrates the PCM repository of a Bookstore application, which is used as a running example in this thesis.
In order to use a PCM instance for performance prediction, the performance-relevant behavior of each service implementation provided by the components must be specified. In this thesis, we will limit ourselves to one supported formalism—the Resource Demanding SEFF (RDSEFF). Similar to activity modeling employing the UML [Object Management Group, Inc., 2013b], an RDSEFF specifies a service implementation as a control flow of actions. PCM distinguishes between internal actions and external call actions—the former being a quantitative specification of the hardware and software resources used by the service; the latter denoting calls to required services. RDSEFFs provide additional features like probabilistic and guarded branches, loops, and operations on variables. Figure 4.8b illustrates the RDSEFF of the Bookstore's *searchBook* service.

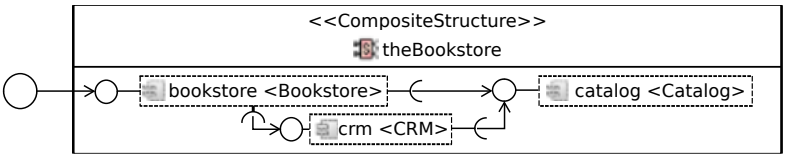


Figure 4.9. PCM system diagram of the Bookstore application

System

A PCM system model provides a deployment-independent component-connector view of the system assembly. Components defined in the repository can be (potentially multiply) instantiated as so-called *assembly contexts* and inter-connected using so-called *assembly connectors*—constrained by the interface providing/requiring specification. The services provided and required by the system are delegated to/from the implementing assembly contexts. Figure 4.9 illustrates the Bookstore's system model.

Resource Environment

A PCM resource environment model specifies the available resource infrastructure and its performance-relevant characteristics. *Resource containers*, e. g., physical servers, are inter-connected by *linking resources*, e. g., network links. Each resource container is associated with the contained processing resources (e. g., CPU and HDD), which can be demanded in the RDSEFFs. For each resource, the resource environment model contains a specification of the performance-relevant properties of the resources—e. g., capacity, processing rates, throughput, and scheduling disciplines.

Allocation

A PCM allocation model specifies the deployment of the system's assembly contexts to resource containers. Each of these mappings is modeled as an *allocation context*.

Usage model

A PCM usage model allows to specify closed and open workloads. Probabilistic user behavior is described in an RDSEFF-like formalism including branches, loops, and calls to system-provided services. Closed workloads include the definition of population size and think time; open workloads include the definition of inter-arrival times.

Part II

SLAstic Approach

Chapter 5

Research Design

This chapter gives an overview of the research design employed for the work on this dissertation. Section 5.1 describes the scope and the vision of this research, and lists the research questions to be addressed. Based on this, Section 5.2 describes the research plan and provides a summary of results. The collaborations that were conducted in the context of this research are outlined in Section 5.3.

5.1 Scope, Vision, and Research Questions

Section 5.1.1 describes the scope and outlines the envisioned approach. The research questions addressed in this research are listed in Section 5.1.2. Note that an introduction to the relevant foundations has previously been provided in Chapters 2 to 4.

5.1.1 Scope and Vision

The scope of this research is the development of a self-adaptive capacity management approach for component-based software systems (CBSSs), employing architecture-based runtime reconfiguration and model-driven techniques. The objective of the adaptation is that the capacity of the controlled software is adapted based on high-level goals, such as meeting SLAs while trying to minimize resource usage. An intuitive strategy is to increase/decrease the provided capacity with increasing/decreasing demand for capacity, particularly based on the workload intensity. We want to cover both reactive and proactive strategies. Considered adaptation actions focus on architectural runtime reconfiguration operations. The decision on when and how to adapt the system are based on architectural models

used at runtime, including QoS-relevant information continuously obtained from the controlled system. Particularly, this includes the possibility to use model-driven techniques for performance evaluation at runtime. MDSE techniques help to increase the degree of automation in the framework, e. g., by automatic system instrumentation.

To summarize, the envisioned approach comprises a combination of architectural modeling languages for CBSSs, an extensible architecture-based self-adaptation framework, architectural runtime reconfiguration for CBSSs, as well as supporting model-driven techniques. The targeted class of CBSSs to be supported are enterprise application systems.

5.1.2 Research Questions

In particular, this thesis addresses the following research questions based on the previously described scope and envisioned approach:

- RQ1: Which aspects need to be modeled?
- RQ2: What is a suitable modeling language?
- RQ3: What are relevant QoS measures to be monitored?
- RQ4: What are basic analyses for online capacity management?
- RQ5: What is a framework that supports the SLAstic approach?
- RQ6: Where and how can MDSE techniques support the approach?
- RQ7: What are suitable reconfiguration operations to control system capacity?

These research questions will be addressed by the activities conducted as part of the work packages described in the following section.

5.2 Research Plan and Summary of Results

Based on the previously described vision and research questions, the research for this thesis is structured into the following five work packages WP1–WP5, whose goals and results will be detailed in the following Sections 5.2.1 to 5.2.5:

- WP1: Architectural Modeling
- WP2: Online Capacity Management Framework
- WP3: Model-Driven Online Capacity Management
- WP4: Runtime Reconfiguration for Controlling Capacity
- WP5: Evaluation

For each work package, we present the goals and a summary of results. We refer to the respective parts in this thesis and refer to other sources for work not being covered or detailed in this document.

The chapter structure in this thesis (also summarized in Section 1.3) reflects the structure of the research plan. The approach is presented in the following Chapters 6 to 10 of this Part II in a sequence matching the order of work packages. The work on utilizing the Palladio Component Model, which is cross-cutting the work packages, is included in Chapter 11 at the end of this part. The evaluation is presented in Part III. Foundations and related work analyzed from literature as part of the research on the work packages are summarized in Part I and in Chapter 16.

In the early phase of this research, the overall vision and the research plan have been presented to the related research communities at different workshops [van Hoorn, 2009a,b; van Hoorn et al., 2009a,b]. Moreover, the research was planned in a proposal document that has been refined together with the PhD supervisor at the beginning of this research in 2008–2009.

5.2.1 WP1: Architectural Modeling

Goals

This work package comprises the activities for selecting or developing a suitable modeling language to be used for the SLAstic approach. Employing the modeling language, it must be able to express various architectural aspects about the controlled software system up to an appropriate level of detail. This includes structural and behavioral information about the component-based architecture, performance requirements (SLAs) and properties, as well as adaptation constraints, policies, etc. A model expressed with the language serves as the basis for monitoring instrumentation, for framework initialization, and for the online analysis at runtime.

Particularly, this work package addresses the research questions RQ1 (*Which aspects need to be modeled?*) and RQ2 (*What is a suitable modeling language?*).

Based on a survey of existing architecture and performance modeling approaches, the expected outcomes of this work package are a specification and an implementation of the selected or developed modeling language, which is usable for the subsequent work packages.

Summary of Results

The main results of this work package are the architectural modeling languages described in Chapter 6 and their integration with the Palladio Component Model (PCM), as described in Chapter 11. With respect to architectural modeling, the core of the SLAstic meta-model is the ability to provide a structural view on CBSS architectures using an abstraction level close to the one provided by PCM. Additionally, the SLAstic meta-model includes concepts to model system behavior and usage, as well as reconfiguration capabilities, operations, and plans. We integrated the meta-model agnostic modeling languages SMM/MAMBA (introduced in Section 4.1.4) and S/T/A (introduced in Section 3.4.3), which are a result of a collaborative research in the context of this thesis (Section 5.3). MAMBA is employed to attach performance measures to SLAstic models. S/T/A is mainly employed to express reconfiguration plans. With respect to the integration of PCM, this includes the transformation of SLAstic models to PCM models and the concept of decorating PCM models by a SLAstic model for use at runtime. Meta-model implementations conforming to the Ecore meta-meta-model are available for all of the developed modeling languages. For the SLAstic meta-model, the Ecore version is an export from Rational Software Architect. Parts of the SLAstic meta-model have additionally been specified in Object-Z, focusing on specific modeling constraints. Based on the M2T transformation provided by EMF, the meta-models were transformed to Java code to make them available for the SLAstic framework implementation.

The research on this work package started with a study of relevant foundations from related research areas, including languages for modeling software architectures (ADLs), performance, and adaptation, as well as meta-modeling concepts and technologies from MDSE (Chapters 2 to 4). Already in an early phase of this research, we came to the conclusion that

the component model (architectural style) and modeling language provided by PCM matches large parts of the requirements for our approach. Initially, we used PCM directly and created a decorator meta-model for adding information on SLAs and reconfiguration constraints and operations to PCM instances [Stöver, 2009]. Thanks to today's MDSE technologies, which ease the development of modeling languages and tool support, we decided to develop the SLAstic meta-model tailored to our approach. As opposed to PCM, the SLAstic meta-model does not aim to be a complete meta-model including the QoS-relevant or performance-relevant information to predict software performance (QoS). Instead, we kept the SLAstic meta-model compact with a focus on runtime reconfiguration. Thanks to the transformation to PCM instances and the decoration concept, it can be used for performance prediction.

5.2.2 WP2: Online Capacity Management Framework

Goals

This work package comprises the activities for developing a self-adaptation framework for the SLAstic approach. Employing the framework, it must be possible to continuously monitor relevant QoS measures from a software system, continuously perform capacity planning at runtime, and to reconfigure the controlled system at runtime. The framework should provide a separation of architecture and technology in that *a*) the online analyses are based on the architectural runtime model (conforming to the modeling language from WP1) and architecture-level QoS measures, and that *b*) it is possible to connect the framework to systems implemented with different technologies. It should provide the basic structure and components for online capacity management, which should be extensible for customization, e. g., w.r.t. algorithms, analyses methods, and supported technologies. In addition to its primary use for *online* capacity management, the framework should also be used for *offline* analysis, e. g., by importing previously monitored data. The purpose of the *offline* operation to be supported by the framework is the ability to evaluate different adaptation strategies, configurations, etc., using pre-recorded workloads. In order to ease this evaluation without having the need for a running system, the integration of a simulator is desirable.

Particularly, this work package addresses the research questions RQ3 (*What are relevant QoS measures to be monitored?*), RQ4 (*What are basic analyses for online capacity management?*), and RQ5 (*What is a framework that supports the SLAstic approach?*).

The basis for this work package is a literature study on state-of-the-art in software performance engineering (including capacity management, performance prediction, etc.) and self-adaptation, including supporting frameworks. The expected outcome of this work package is a conceptual architecture and a proof-of-concept implementation of a framework supporting the SLAstic approach. With respect to instrumentation and dynamic analysis of Java EE-based enterprise application systems, we can build on existing work developed in our research group [Focke, 2006; van Hoorn, 2007; Rohr et al., 2008]. However, in its current state, this work is not supporting online analysis and is not designed for extensibility, as required by our approach.

Summary of Results

The main results of this work package are the Kieker and SLAstic frameworks whose conceptual architectures and implementations are described in Chapters 7 and 8. Kieker provides an extensible framework for implementation-level instrumentation, monitoring, and dynamic analysis of software systems. Kieker became a big focus of this thesis and resulted in a framework that supports but is not limited to the SLAstic approach. Also thanks to enormous contributions by colleagues (see also Sections 5.3 and 15.2.4), its implementation became way more than a proof of concept. Building on Kieker, the SLAstic framework provides a reusable and extensible platform for architecture-based online capacity management. The framework implements a MAPE-K-based self-adaptation loop (Section 3.4.1). The framework is tailored to online capacity management by the components for *a*) performance evaluation, *b*) workload forecasting, *c*) performance prediction, and *d*) adaptation planning, which are contained in the framework's adaptation controller component. A core component is a model manager that maintains a (continuously updated) runtime model conforming to the SLAstic metamodel developed in WP1. Online and offline analysis modes are supported. The SLAstic.SIM discrete-event simulator for runtime reconfigurable PCM instances has been developed and integrated (Section 11.3). SLAstic builds

on Kieker in that it uses it for collecting and providing runtime data from the simulated systems and for importing previously recorded workload stimuli.

The research on this work package started with a study of relevant foundations from SPE and self-adaptation (Chapter 4 and Section 3.4) in combination with the work on the foundations for WP1. Stöver [2009] developed a first version of the SLAstic framework (Section 5.3.1), which has been further refined in the future time based on the experiences. The initial framework version used PCM as a runtime model. During SLAstic's development, various parts that are useful for dynamic software analysis in general were integrated into the Kieker framework. The development of SLAstic.SIM, our discrete-even simulator for runtime reconfigurable PCM instances, was performed in the context of the thesis by von Massow [2010]. Work on prediction of performance measures based on time series analysis has been performed in the context of the research on the ΘPAD approach [Bielefeld, 2012; Frotscher, 2013] (Section 5.3). Work on the use of model repository technologies, e. g., to be used within the model manager contained in the SLAstic framework, was performed in the context of the thesis by Kiel [2013] (Section 5.3.1). Chapter 15 provides a retrospective look at Kieker's history, development, and impact.

5.2.3 WP3: Model-Driven Online Capacity Management

Goals

This work package comprises the activities to improve automation of reoccurring, schematic tasks within the SLAstic approach employing model-driven techniques—particularly model transformations—and technologies from MDSE. Potential tasks to be automated are the generation of measurement instrumentation, the transformation of implementation-level measurements to architecture-level QoS measures, extraction of SLAstic models from implementation-level measurement data, and model-driven analyses at runtime.

Particularly, this work package addresses the research question RQ6 (*Where and how can MDSE techniques support the approach?*).

The expected outcome of this work package is a set of techniques for the afore-mentioned automation using model-driven techniques. The

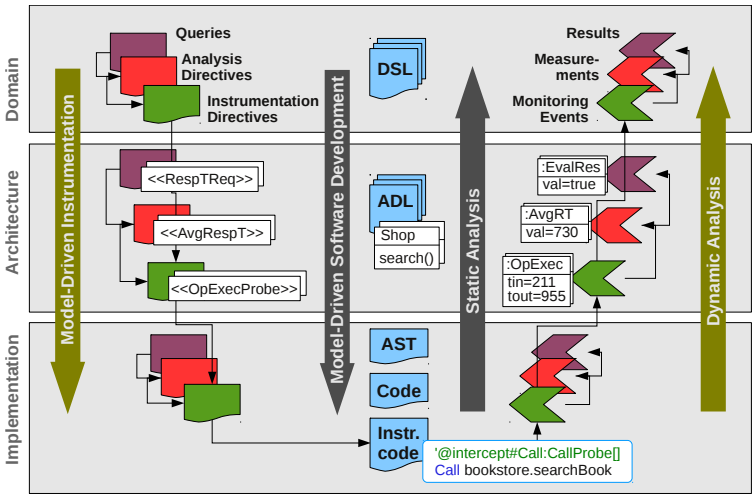


Figure 5.1. Model-driven instrumentation and analysis in the DynaMod approach [van Hoorn et al., 2011a]

techniques should be implemented as a proof-of-concept. Most likely, the activities require extensions to the modeling languages developed in WP1.

Summary of Results

The main results of this work package are the model-driven techniques for the SLAstic framework, described in Chapter 9, and integrating PCM as described in Chapter 11. With respect to the results described in Chapter 9, this particularly includes the following three aspects: *a*) model-driven generation of a Kieker instrumentation based on modeling constructs contained in the SLAstic meta-model and MAMBA; *b*) transformation of implementation-level Kieker records into SLAstic monitoring events; *c*) extraction and updates of a SLAstic model based on Kieker records and SLAstic monitoring events. As described in Chapter 7, Kieker also provides model extraction functionality based on dynamic analysis. Kieker includes an earlier and rudimentary version of the SLAstic meta-model. With respect to the PCM integration, the results for this work package include *a*) the transforma-

tion from SLAstic models to PCM instances, which—together with the afore-mentioned extraction of SLAstic models—enables a basic extraction of PCM instances via dynamic analysis for further refinement; *b*) the decoration of PCM instances, which enables the use of PCM-based performance prediction at runtime.

In the DynaMod research project [van Hoorn et al., 2011a], we already built on the approach for model-driven instrumentation and analysis developed for this thesis—combining it with static analysis. Figure 5.1 provides an illustration of the approach from our publication on this topic [van Hoorn et al., 2011b]. In order to make the modeling foundations meta-model agnostic and automate the interpretation of measurements, we started a joint work (Section 5.3) on the MAMBA framework, which builds on the OMG meta-model specification SMM, as detailed in Sections 2.3.1 and 4.1.4.

5.2.4 WP4: Runtime Reconfiguration for Controlling Capacity

Goals

This work packages comprises the activities to select a set of architectural runtime reconfiguration operations that serve to control the capacity of component-based software systems and integrate these operations into the SLAstic approach. The goal is to use these operations in our approach to increase the system's resource efficiency while meeting SLAs.

Particularly, this work package addresses the research question RQ7 (*What are suitable reconfiguration operations to control system capacity?*).

The expected outcome of this work package is the selection, specification, integration, and proof-of-concept implementation of the considered operations. The basis for this work package is a study of architectural runtime reconfiguration approaches. We can build on existing work on transparent runtime reconfiguration of component-based software systems, conducted by Matevska [2009].

Summary of Results

The main results of this work package are the five architectural runtime reconfiguration operations considered in this thesis and their integration

into the framework (including S/T/A), as described in Chapter 10. With respect to a specification and proof-of-concept implementation, we defined the operation semantics for the Palladio Component Model and implemented them in SLAstic.SIM (Section 11.3). In our lab experiments (WP5), these architectural reconfigurations were implemented for Java EE and IaaS technologies as additional proofs of concept (Chapter 13).

An important basis for the results in this work package was the work by Matevska [2009] on transparent reconfiguration of component-based systems. We jointly supervised the thesis by Bunge [2008] who implemented a transparent runtime reconfiguration in a Java EE application server and performed quantitative experiments, e. g., w.r.t. the costs of reconfiguration (see also Section 5.3).

5.2.5 WP5: Evaluation

This work package comprises all activities related to the goal of evaluating the overall approach including its components developed in WP1–4.

Table 5.1 provides an aggregated view on the evaluation methodology including the evaluation questions and measures as well as the associated evaluation methods and scales of measurement. Roughly based on the Goal Question Metric (GQM) approach [Basili et al., 1994; van Solingen and Berghout, 1999], we define a set of (evaluation) *questions*. With each question, we associate a number of *measures* to be obtained by applying one or more evaluation methods.[1] We use a combination of the following evaluation methods: *a*) proof-of-concept implementation, *b*) case study, *c*) lab experiment, *d*) simulation, *e*) literature review, and *f*) argumentation. For a measure we distinguish *qualitative and quantitative scales of measurement*. Qualitative measures employ a scale of measurement that is *nominal* or *ordinal*. As opposed to that, quantitative measures use interval or ratio scales. For an introduction into scales (or levels) of measurement, please refer to respective literature, e. g., by Zuse [1998], and by Field and Hole [2012].

The decision to employ a combination of the afore-mentioned evaluation methods is that we are convinced that their complementary use is well-suited to answer the considered questions.

[1] As discussed in Section 4.1.1, we favor the notion of *measure* over *metric*—even though, the latter is used in the GQM context.

Table 5.1. Evaluation questions (EQ), measures (EM), methods, and scales of measurement

		Method						Scale	
		Proof of concept	Case study	Lab experiment	Simulation	Literature review	Argumentation	Qualitative	Quantitative
EQ1:	Is the overall approach applicable to realistic scenarios?								
	EM1.1: Confirmation of assumptions								
	EM1.1.1: Variations in workload		●			●			●
	EM1.1.2: Underutilized resources		●			●			●
	EM1.2: Perturbation by application monitoring		●	●				●	○
	EM1.3: Suitability of modeling language	●	●	●	●			●	
	EM1.4: Suitability of separating architecture and technology	●	●	●	●			●	
EQ2:	Does the approach have the desired properties?								
	EM2.1: Extensibility of framework for specific purposes and technologies								
	EM2.1.1: Modeling language						●	●	
	EM2.1.2: Monitoring	●	●	●	●			●	
	EM2.1.3: Analysis	●	●	●	●			●	
	EM2.1.4: Reconfiguration	●		●	●			●	
	EM2.2: Reusability of framework								
	EM2.2.1: Modeling language	●	●	●	●			●	
	EM2.2.2: Monitoring	●	●	●	●			●	
	EM2.2.3: Analysis	●	●	●	●			●	
	EM2.2.4: Reconfiguration	●		●	●			●	
	EM2.3: Suitability of reconfiguration operations								
	EM2.3.1: Impact on Capacity			●	●				●
	EM2.3.2: Transparency			●	●				●
	EM2.4: Suitability of MDSE techniques	●	●	●	●			●	
EQ3:	How does the approach compare to other approaches?								
	EM3.1: Novelty					●	●	●	
	EM3.2: Validity and performance				●				●
EQ4:	How do we assess our work?								
	EM4.1: Degree of reaching the goals						●	●	
	EM4.2: Impact (use by others)					●	●	●	

- Proof-of-concept implementations serve to show the applicability of approaches with respect to technical feasibility, and enable their use in other experimental evaluation methods.
- Case studies, focusing on industrial enterprise application systems, are a valuable means for a variety of reasons, e. g., for getting feedback on the developed approach, gaining knowledge about state-of-the art technologies and best practices (e. g., w.r.t. system architectures, use of MDSE, and monitoring), as well as to have access to production workload and performance data, which may contribute to confirming assumptions underlying this research (varying usage profiles, underutilized resources, etc.).
- Lab experiments serve to obtain evaluation results in controlled environments. The envisioned experimental setup comprises the use of the developed adaptation framework with a sample application exposed to synthetic workload—ideally based on production usage profiles.
- Particularly with respect to the evaluation of reconfiguration operations, simulation serves as a more flexible method compared to lab experiments, because it is not necessary to implement the reconfiguration operations.
- Literature review is used to identify, build on, and compare previous approaches and results by other researchers.
- For questions that are not answered by either of the aforementioned methods, we use the method of argumentation. The respective question is then answered by a discussion which is mainly based on a subjective assessment rather than on qualitative or quantitative results obtained from experiments.

Proof-of-concept implementations have been developed for the SLAstic meta-model including the S/T/A and SMM/MAMBA languages and their integration (WP1), the Kieker and SLAstic frameworks (WP2), the supporting model-driven techniques (WP3), the five runtime reconfiguration operations for controlling capacity and their integration into the SLAstic approach (WP4), as well as the integration of PCM. These proof-of-concept implementations form the basis for three types of experimental evaluations, namely an industrial case study (Chapter 12), lab experiments (Chapter 13), as well as simulation (Chapter 14). In these experimental evaluations, three types of systems were used in online and offline settings: a distributed

Java-based web portal in production (case study), a Java-based sample application in a private cloud setting (lab experiment), and the running Bookstore example used throughout the thesis (simulation). The evaluation results can be found in Chapters 12 to 16, as well as in Chapter 17. The conducted non-experimental evaluations comprise a retrospective view on Kieker's history, development, and impact (Chapter 15), as well as a discussion of related work (Chapter 16).

5.3 Collaborations in the Context of this Research

This section summarizes the collaborations with students (Section 5.3.1) as well as with researchers and industry (Section 5.3.2) in the context of this thesis.

5.3.1 Students

A considerable number of students helped in conducting the research for this thesis as part of Master's and Diploma theses,[2] seminars, and student projects. These works were conducted at the Universities in Kiel and Oldenburg and have been partially co-supervised with the following researchers and partners from collaborating companies: Sören Frey, Wilhelm Hasselbring, Reiner Jung, Stefan Kaes (XING), Holger Knoche (b+m Informatik), Jasminka Matevska, Matthias Rohr, Thomas Stahl (b+m Informatik), and Jan Waller.

In this section, we will briefly summarize the core works that contributed to this thesis in ascending chronological order. Note that details on a couple of these works are not further detailed in this thesis.

- Bunge [2008] contributed to WP4 (Section 5.2.4) by providing a proof-of-concept implementation of a runtime reconfiguration operation in a Java EE application server, including a quantitative evaluation in lab experiments with respect to the following measures while executing reconfigurations: failure rates, user-perceived response times, CPU utilization.

[2] At German universities, a Diploma thesis (Diplomarbeit) is a 6-month final thesis for the Diploma degree (Diplom), which is equivalent to a Master's degree.

- Stöver [2009] contributed to WP1 (Section 5.2.1) and WP2 (Section 5.2.2). With respect to WP2, she developed an initial version of the online adaptation framework, including a qualitative and quantitative evaluation based on lab experiments. With respect to WP1, Stöver used PCM as the runtime model, extending it by meta-models (including a textual concrete syntax) for reconfiguration specifications, reconfiguration plans, and SLAs.

- von Massow [2010] contributed to WP2 (Section 5.2.2) and WP4 (Section 5.2.4). He developed the SLAstic.SIM simulator for runtime reconfigurable PCM models, which can be connected the SLAstic framework. With respect to WP4 he contributed a proof-of-concept implementation of the reconfiguration operations considered in this thesis.

- As part of a Master's seminar and his Master's thesis (not supervised by me), Fittkau [2011, 2012] contributed to WP2 (Section 5.2.2) and WP5 (Section 5.2.5) by developing adapters for the SLAstic online adaptation framework to the cloud computing platforms Eucalyptus and Amazon Web Services (AWS) (Section 3.3.2), and using the framework in different lab experiments.

- Magedanz [2011] contributed to WP2 (Section 5.2.2) and WP5 (Section 5.2.5) by developing a solution for Kieker to monitor .NET applications and working on the Nordic Analytics case study, mentioned in Section 15.3.1.

- Günther [2011] contributed to WP3 (Section 5.2.3) by working on the transformations of SLAstic models into PCM models.

- Bielefeld [2012] contributed to WP2 (Section 5.2.2) and WP5 (Section 5.2.5) by working on forecasting support based on time series analysis and the XING case study. The same holds for Frotscher [2013], who extended Bielefeld's work.

- Richter [2012] contributed to WP2 (Section 5.2.2), WP3 (Section 5.2.3), and WP5 (Section 5.2.5) by developing a generative method for instrumenting COBOL applications.

- Kiel [2013] contributed to WP2 (Section 5.2.2) by evaluating the scalability of model repository technologies, especially CDO, which can be used to manage the models at runtime. Moreover, Kiel implemented considerable parts of the MAMBA framework.

5.3.2 Researchers and Industry

This research included various collaborations with researchers from our and other research groups, which also contributed to this work. With respect to Kieker (WP2, Section 5.2.2), these collaborations were conducted with many colleagues from the University of Oldenburg and Kiel University, leading to joint publications (e. g., [Rohr et al., 2008; van Hoorn et al., 2009c, 2012]). MAMBA (WP1 and WP3, Sections 5.2.1 and 5.2.3) was joint work with colleagues from Kiel University [Frey et al., 2011, 2012], also in the context of other research projects. With respect to transparent runtime reconfiguration of CBSSs (WP4), we worked together with [Matevska, 2009], e. g., as part of the joint supervision of Bunge's thesis. S/T/A (WP1 and WP2, Sections 5.2.1 and 5.2.2) was joint work with colleagues from the Descartes research group at the Karlsruhe Institute of Technology (KIT) [Huber et al., 2012, 2014]. We also collaborated with this research group on workload forecasting in the context of Herbst's and Bielefeld's Master's theses [Herbst, 2012; Bielefeld, 2012; Herbst et al., 2013a]. Also, mainly in the Kieker context, we worked together with a number of companies, as detailed in Section 15.3.1.

Chapter 6

Architectural Modeling

This chapter describes the SLAstic meta-model that has been developed for being used in the SLAstic approach for representing architectural information about the controlled component-based software system (CBSS) and the adaptation process. The meta-model builds the basis for other parts of the SLAstic approach, detailed in Chapters 8 to 11. For example, instances of the SLAstic are used at runtime by the SLAstic online adaptation framework (Chapter 8) and by the supporting MDSE techniques (Chapter 9), e. g., for model-driven instrumentation. The meta-model integrates with the complementary meta-model agnostic modeling approaches S/T/A and MAMBA, which were developed in the context of this thesis and have been described in Section 3.4.3 and Section 4.1.4 respectively. All meta-model implementations are available as part of the supplementary material to this thesis [van Hoorn, 2014].

The remainder of this chapter is structured as follows. Section 6.1 introduces the languages and technologies used for the specification and implementation of the SLAstic meta-model. The subsequent sections detail the SLAstic meta-model based on the organization into aspects concerning system structure (Section 6.2), behavior and usage (Section 6.3), adaptation and reconfiguration (Section 6.4), as well as QoS measures (Section 6.5).

6.1 Specification Languages and Implementation

This chapter uses a combination of *a*) the Unified Modeling Language (UML) 2.5 [Object Management Group, Inc., 2013b] and *b*) the Object-Z specification language [Smith, 2000] to represent the meta-models. We follow the common approach of employing UML class diagrams for visualizing the abstract syntax of the meta-model (Chapter 2).

As the primary reference, the SLAstic meta-model is implemented with Rational Software Architect (RSA), a well-known MDSE tool. The RSA-based meta-model representation is exported to Ecore, in order to make it available to the EMF-based MDSE tooling infrastructure employed in this thesis, including automatic generation of Java code (Section 2.3.2). We additionally specified a subset of the SLAstic meta-model in Object-Z, mainly to formalize additional constraints on the meta-model. We prefer Object-Z to OCL for this purpose due to Object-Z's more compact representation. Basic validations, like type checking, on the Object-Z were performed using publicly available tools [Community Z Tools Project, 2014]. For a comprehensive introduction into Object-Z and the underlying Z Specification Language, please refer to other sources, e. g., by Smith [2000], Woodcock and Davies [1996], as well as Spivey [2001]. The SMM/MAMBA and S/T/A meta-models (as described in Sections 3.4.3 and 4.1.4) are specified directly in Ecore.

Note that the descriptions of the modeling languages focuses on the core concepts and omits unimportant details. The implementations of the modeling languages serve as the main reference [van Hoorn, 2014].

6.2 System Structure

A SLAstic system model is partitioned into four complementing sub-models, each of which provides a specific architectural view on the modeled system: The *type repository* model specifies the set of available software component types along with information on their required and provided interfaces, as well as available types of execution containers, hardware and software resources, etc. As a logical view on the software composition, the *component assembly* model specifies the set of assembly components —being instances of the component types from the type repository—as well as their inter-connection via connectors. The set of available execution containers, along with information on their inter-connection via network links, is specified in the *execution environment* model. The *component deployment* model specifies the mapping of the assembly components from the component assembly model to execution containers, as so-called deployment components.

The partitioning of the system model into the four submodels is depicted as an Object-Z specification in Figure 6.1. By presenting relevant excerpts of the SLAstic meta-model, the remainder of this section describes these four sub-models more formally.

SystemModel

$$
\begin{array}{l}
typeRepository : TypeRepositoryModel \\
componentAssembly : ComponentAssemblyModel© \\
executionEnvironment : ExecutionEnvironmentModel© \\
componentDeployment : ComponentDeploymentModel© \\
\hline
\forall ac : AssemblyComponent \mid ac \in componentAssembly.assemblyComponents \\
\quad \bullet\ (\exists dc : DeploymentComponent \mid dc \in componentDeployment.deploymentComponents \\
\qquad \bullet\ dc.assemblyComponent = ac)
\end{array}
$$

Figure 6.1. Object-Z specification of the `SystemModel` meta-class referencing the four submodels (© denotes object containment). The specification includes the constraint that each assembly component must have at least one corresponding deployment component.

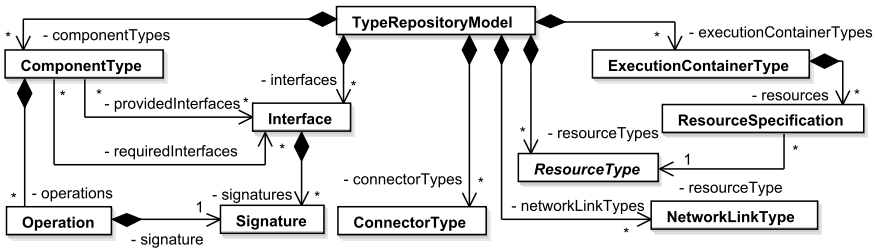


Figure 6.2. Subset of the meta-classes for the type repository. Note that attributes are not shown.

6.2.1 Type Repository

A type repository model (`TypeRepositoryModel`) contains type specifications of components (`ComponentType`), interfaces (`Interface`), connectors (`ConnectorType`), execution containers (`ExecutionContainerType`), resources (`ResourceType`), and network links (`NetworkLinkType`), detailed in the remainder of this section. Note that we will exclude connector and network link types from the following description as they have not been further considered in this thesis. Relevant parts of the meta-model related to the type repository model are depicted in Figure 6.2.

ComponentType
FQNamedElement

$$
\begin{array}{l}
providedInterfaces : \mathbb{P}\, Interface \\
requiredInterfaces : \mathbb{P}\, Interface \\
operations : \mathbb{P}\, Operation© \\
\hline
\forall o_1, o_2 : Operation \mid (o_1 \neq o_2 \land \{o_1, o_2\} \subseteq operations) \bullet o_1.signature \neq o_2.signature \\
\{s : Signature;\ i : Interface \mid i \in providedInterfaces \land s \in i.signatures \bullet s\} \\
\qquad = \{s : Signature;\ o : Operation \mid o \in operations \land s = o.signature \bullet s\}
\end{array}
$$

Figure 6.3. Object-Z specification of the `ComponentType` meta-class, including the constraint that a `ComponentType` must have an `Operation` for each `Signature` of the provided `Interface`s. By extending the meta-class `FQNamedElement`, component types have a fully-qualified name. $\mathbb{P}$ denotes the power set, which corresponds to a reference with multiplicity "many" (e. g., $*$) in class diagrams.

▷ *Component Types, Interfaces*

Interfaces declare a set of signatures (`Signature`) having a name, a list of parameter types, and a return type as attributes. A component type declares provided and required interfaces, and implements a set of operations (`Operation`) with distinct signatures. For each interface provided by a component type, a corresponding operation (`Operation`) with an equal signature must be implemented by that component type (formalized in Figure 6.3). Internal modeling of operation implementations is not part of the type repository. The usage profile (Section 6.3.3) includes quantitative information on calling relationships between operations and signatures of required interfaces. Note that we currently omit type inheritance, i. e., we assume a flat type hierarchy for all elements contained in the type repository.

▷ *Execution Container Types, Resource Types, and Resource Specifications*

An execution container type includes a specification of which hardware resources each execution container (Section 6.2.3) of this type contains. Resources may be of different types, modeled by the abstract class `ResourceType`. Concrete resource type classes exist for CPU (`CPUType`) and

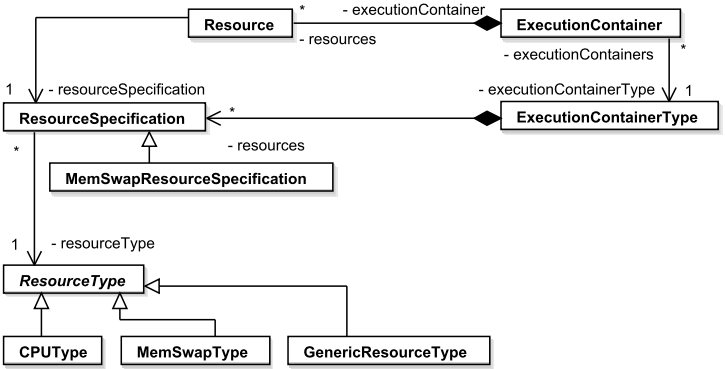
memory (*MemSwapType*) resources. *CPUType* includes attributes for vendor, model, and clock rate. In certain cases, it is sufficient to use the generic resource type (*GenericResourceType*). Contained in an execution container type are resource specifications (*ResourceSpecification*), which instantiate resource types for an execution container type. For example, an execution container type may be equipped with multiple CPUs of the same resource type. For memory resources, a specific resource specification exists, which includes instance-specific information on the resource (size of memory and swap). Figure 6.4 includes the specifications of the concrete meta-classes for resource types and resource specifications.

6.2.2 Component Assembly

A component assembly model (*ComponentAssemblyModel*) provides a view on a system's logical assembly in terms of assembly components (*AssemblyComponent*) and their interconnection using assembly connectors (*AssemblyConnector*). Both assembly components and assembly connectors are typed with respect to the component and connector types defined in the type repository model (Section 6.2.1). As for component types, a system assembly declares a set of required and provided interfaces. Delegation connectors are used to delegate calls directed to provided interfaces further to assembly components that provide an implementation for these interfaces; likewise, required interfaces of assembly components may be connected to required interfaces of the component assembly for delegation. The meta-modeling concepts and constraints related to component assembly are included in Figure 6.5 (on page 97).

6.2.3 Execution Environment

An execution environment model (*ExecutionEnvironmentModel*) specifies the set of available execution containers (*ExecutionContainer*) as well as their interconnection via network links (*NetworkLink*). According to the component assembly specification described in the previous Section 6.2.2, the execution containers and network links are typed based on the respective types defined in the type repository. Figure 6.6 (page 98) shows the relevant parts of the concepts related to the execution environment that have been described so far. Figure 6.4 includes the parts of the meta-model relevant to



(a) Abstract and concrete meta-classes for resource modeling and their use in execution containers and execution container types

CPUType
ResourceType

vendor : *String*
model : *String*
clockRateMhz : $\mathbb{N}$

(b) Meta-class for CPU resource type

MemSwapResourceSpecification
ResourceSpecification

memCapacityBytes : $\mathbb{N}$
swapCapacityBytes : $\mathbb{N}$

resourceType $\in \downarrow$*MemSwapType*

(c) Meta-class for memory-specific resource specification

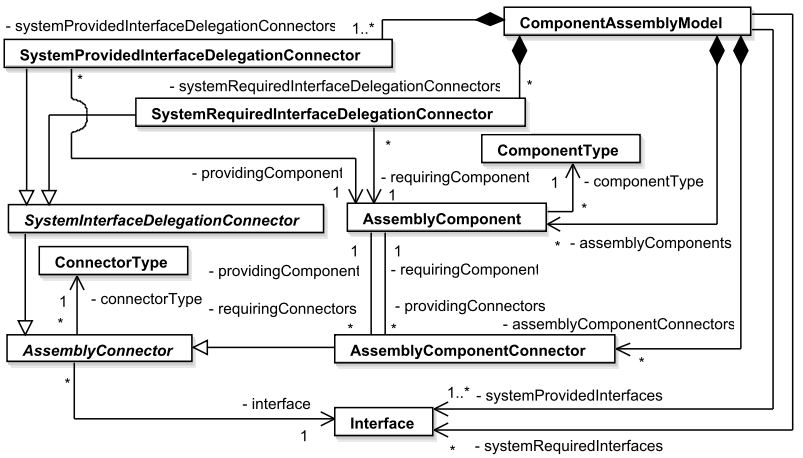
ExecutionContainer

executionContainerType : *ExecutionContainerType*
networkLinks : $\mathbb{P}$ *NetworkLink*
resources : $\mathbb{P}$ *Resource*©

$\forall$ *resSpec* : *ResourceSpecification* $\bullet$
(*resSpec* $\in$ *executionContainerType.resources*
$\Leftrightarrow$ ($\exists$ *res* : *Resource* | *res* $\in$ *resources* $\bullet$ *res.resourceSpecification* = *resSpec*))

(d) Meta-class for execution containers including the constraint that each container's resources must match its type's resource specifications

Figure 6.4. Object-Z specifications of meta-classes for resource types and resource specifications. $\downarrow$ denotes the union of a type and all of its subtypes.



(a) Subset of the meta-classes for the component assembly

AssemblyComponent

$$
\begin{array}{l}
componentType : ComponentType \\
providingConnectors : \mathbb{P}\, AssemblyConnector \\
requiringConnectors : \mathbb{P}\, AssemblyConnector \\
\hline
componentType.providedInterfaces = \\
\quad \{i : Interface;\ c : AssemblyConnector \mid c \in providingConnectors \land c.interface = i \bullet i\} \\
componentType.requiredInterfaces = \\
\quad \{i : Interface;\ c : AssemblyConnector \mid c \in requiringConnectors \land c.interface = i \bullet i\}
\end{array}
$$

(b) Meta-class `AssemblyComponent` with the constraint that all provided and required interfaces must be connected

AssemblyComponentConnector
AssemblyConnector

$$
\begin{array}{l}
providingComponent : AssemblyComponent \\
requiringComponent : AssemblyComponent \\
\hline
interface \in providingComponent.componentType.providedInterfaces \\
interface \in requiringComponent.componentType.requiredInterfaces
\end{array}
$$

(c) Meta-class `AssemblyComponentConnector` with the constraint that the connected components provide/require the connector's interface

Figure 6.5. Meta-model excerpts of the component assembly model

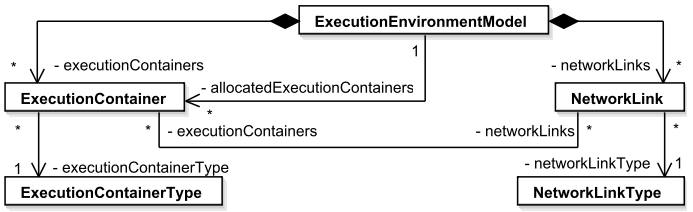


Figure 6.6. Core meta-classes and relations for the execution environment model

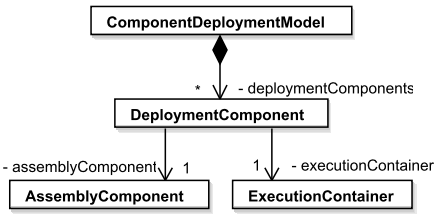


Figure 6.7. Component deployment model containing deployment components

modeling the relation between an execution container's resources and the afore-mentioned resource types and specifications.

6.2.4 Component Deployment

A component deployment model (`ComponentDeploymentModel`) specifies the deployment of assembly components to execution containers, as so-called deployment components (`DeploymentComponent`). Figure 6.7 shows the relevant part of the meta-model. At least one deployment component must exist for each assembly component (included as constraint in Figure 6.1 on page 93).

6.3 System Behavior and Usage

The SLAstic meta-model includes an extensible set of meta-classes for representing basic monitoring events related to the structural meta-model entities—e. g., executions of operations—and information derived from the basic events, such as traces as well as call frequencies and call relationships

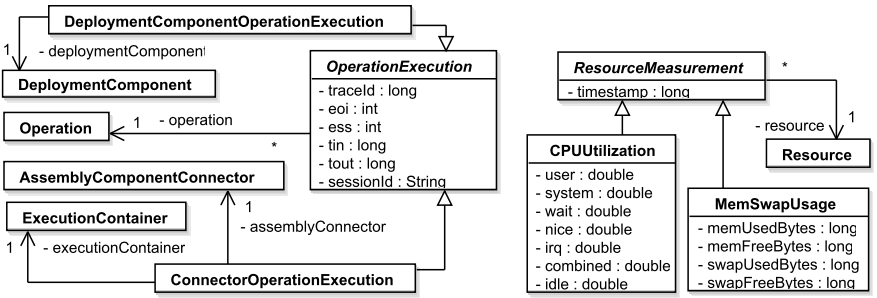


Figure 6.8. Meta-classes for operation execution and resource usage events

for operations. The remainder of this section details the meta-model concepts for monitoring events (Section 6.3.1), traces (Section 6.3.2), and usage (Section 6.3.3).

6.3.1 Monitoring Events

Based on the abstract meta-class `Event`, the SLAstic meta-model currently includes meta-classes for representing monitoring events with respect to executions of operations and usage of resources. Figure 6.8 shows the relevant meta-classes. Note that the current set of meta-classes for monitoring events is tailored to the needs for this thesis. Additional meta-classes can be added in case these are needed for certain analyses.

▻ *Operation Executions*

Each operation execution event (abstract meta-class `OperationExecution`) includes information about trace and session identifiers, the start and end of the operation execution (*tin*, *tout*), as well as information on the position of the execution in a trace (*eoi*, *ess*). We further distinguish between executions of deployment component operations (`DeploymentComponentOperationExecution`) and connector operations (`ConnectorOperationExecution`). Each of these concrete meta-classes adds additional references to respective entities in the SLAstic meta-model.

▻ *Resource Usage*

Each resource usage event (abstract meta-class `ResourceMeasurement`) includes a timestamp of measurement and a reference to the measured resource from the SLAstic meta-model. We further distinguish between usage events for the afore-mentioned resource types, i. e., generic (`ResourceUtilization`), CPU (`CPUUtilization`), and memory (`MemSwapUsage`). The generic event simply includes a single utilization value, while the events for the more concrete resource types include more detailed information known from system measurements, as depicted in Figure 6.8.

6.3.2 Traces

Figure 6.9 shows the meta-classes and relationships for representing traces (abstract meta-class `Trace`). We distinguish two trace representations, which contain equivalent information for valid traces (abstract meta-class `ValidTrace`) and can be transformed into the respective other form: *a*) execution traces (`ExecutionTrace`) represent a trace as a sequence of the afore-mentioned operation executions; *b*) message traces (`MessageTrace`) represent a trace based on the messages interchanged among architectural entities—currently limited to synchronous call and reply messages.

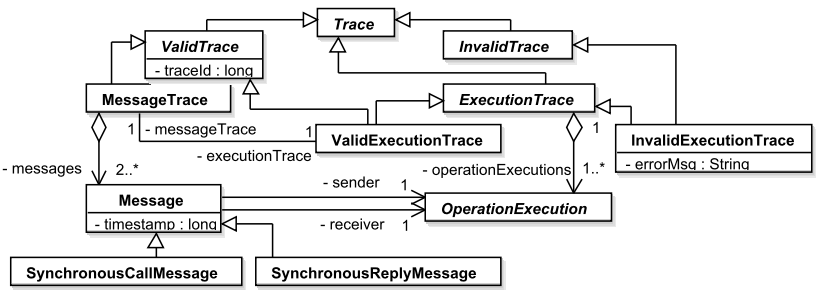


Figure 6.9. Meta-classes and relationships for representing traces

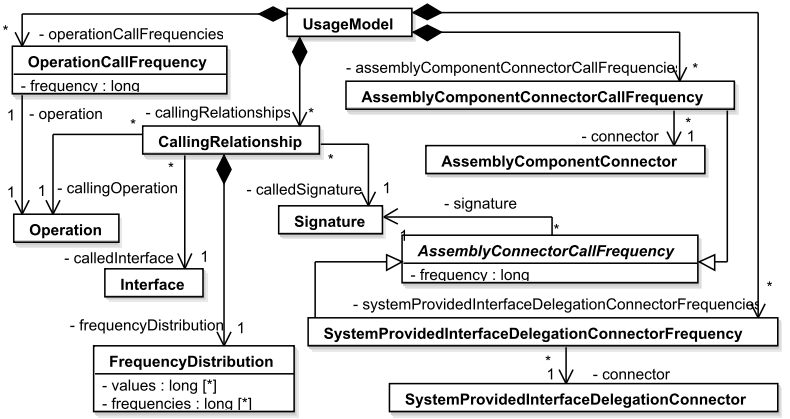


Figure 6.10. Usage model

6.3.3 Usage

The SLAstic usage model contains quantitative information about internal and external calls to operations and interface signatures. The meta-class and relationships are depicted in Figure 6.10. A calling relationship (`CallingRelationship`) stores a frequency distribution (`FrequencyDistribution`) —the information often visualized in histograms—for calls from an operation contained in a component type to a signature of an interface. Three different types of call frequencies (`OperationCallFrequency`, `AssemblyComponentConnectorCallFrequency`, `SystemProvidedInterfaceDelegationConnectorFrequency`) allow to decorate operations of component types, as well as signatures of assembly component connectors and system-provided interface delegation connectors by the information about how many times they have been called during an observation period. Note that similar as for the monitoring events, the usage model meta-model has been developed to support the needs for the analyses in this thesis and can be extended as needed.

6.4 Reconfiguration

The SLAstic meta-model provides an extensible set of concepts for architectural information concerning reconfiguration. Particularly, this includes modeling *adaptation operations*, *adaptation capabilities*, as well as *adaptation plans* and *adaptation results*. In this chapter, we use the notion *reconfiguration* instead of *adaptation* due to the focus of this thesis. We see reconfiguration as a specific type of adaptation denoting the change of an architectural configuration. The described concepts may be used for other types of adaptation as well.

In Section 3.4.3 we have already described the meta-model agnostic S/T/A approach for modeling adaptation processes, which is a result of joint work during the course of this thesis. To summarize, S/T/A allows to model adaptation strategies, tactics, and actions. We will not further cover adaptation strategies in this section but assume that they can be specified using S/T/A. S/T/A formalisms for modeling tactics and actions are integrated in the SLAstic for modeling reconfiguration plans.

Note that many of the concepts introduced in this section are abstract in order to provide the basis for concrete implementations, e. g., targeting runtime reconfiguration for controlling capacity as described in Chapter 10.

6.4.1 Reconfiguration Plan and Reconfiguration Actions

A *reconfiguration operation* defines a parameterized method to modify an architectural configuration. Similar to operations in programming languages, a reconfiguration operation is specified by a signature—including input and output parameters—and semantics that specify the effect of change, e. g., in an imperative or functional way. An execution of a reconfiguration operation is denoted as a *reconfiguration action* (abstract meta-class `ReconfigurationAction` in Figure 6.11). The SLAstic meta-model does not include a dedicated meta-class for reconfiguration operations. Instead, concrete meta-classes extending `ReconfigurationAction` can be considered reconfiguration operations. Parameters are specified as meta-class attributes or references. A subset of the parameters typically refers to meta-classes of the SLAstic meta-model.

An *adaptation plan* groups a set of associated—and possibly inter-dependent—reconfiguration actions. Coming back to the analogy of programming

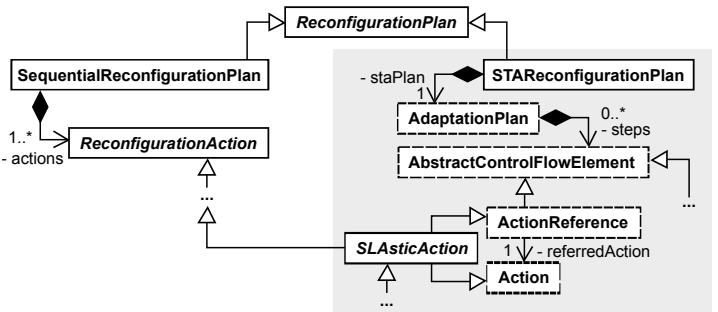


Figure 6.11. Reconfiguration plan including reconfiguration actions. The integration of S/T/A adaptation plans is highlighted with a gray background. Meta-classes from the S/T/A language are depicted with dashed borders.

languages, this refers to a program determining the sequence of statements to be executed, including control flow elements and calls to operations. The abstract meta-class *ReconfigurationPlan* is the basis for an extensible set of reconfiguration plan types. Note that reconfiguration plans are typically created on the fly and, hence, are not contained by another meta-class. Figure 6.11 includes two types, namely *SequentialReconfigurationPlan* and *STAReconfigurationPlan*:

- *SequentialReconfigurationPlan* is a basic reconfiguration plan type that allows to specify a sequence of reconfiguration operations to be executed. One core limitation of this type is that results of one operation execution cannot be passed to subsequent operation calls.
- For being able to express more advanced reconfiguration plans, we integrate the concepts for modeling S/T/A adaptation plans using the *STAReconfigurationPlan* type. Such plan contains an S/T/A adaptation plan (*AdaptationPlan*). As described in Section 3.4.3, an S/T/A adaptation plan includes a set of nested control flow elements (*AbstractControlFlowElement*). The reconfiguration actions from the SLAstic meta-model are integrated by the abstract meta-class *SLAsticAction*. According to the specialization of *ReconfigurationActions*, *SLAsticActions* need to be specialized by extending the respective reconfiguration action meta-class. The reason that *SLAsticAction* extends both *ActionReference* and *Action* is that the

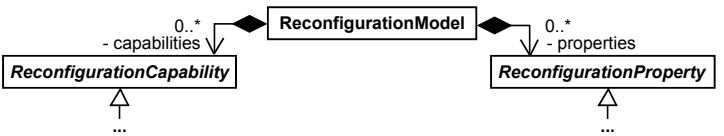


Figure 6.12. Reconfiguration model including reconfiguration capabilities and properties

definition of reconfiguration operations in S/T/A is done on instance level (`Action`) rather than on type level (extension of `ReconfigurationAction` in the SLAstic meta-model).

6.4.2 Reconfiguration Capabilities and Properties

For an adaptable software system, it cannot be assumed that the supported reconfiguration operations can be applied to each architectural entity that—according to reconfiguration operation's signature—supports this reconfiguration in principle. Moreover, additional constraints may exist. In the SLAstic meta-model these aspects are handled under the notion of *reconfiguration capabilities*. Moreover, it can be relevant to model properties of executed reconfiguration operations. As depicted in Figure 6.12, the SLAstic meta-model provides basic support for modeling reconfiguration capabilities (`ReconfigurationCapability`) and reconfiguration properties (`ReconfigurationProperty`), contained in a reconfiguration model (`ReconfigurationModel`). The abstract meta-classes need to be extended based on the specific needs for the respective reconfiguration operations.

6.5 QoS Measures and Instrumentation

The SLAstic meta-model includes concepts to annotate the architectural entities in a SLAstic model by directives for desired QoS measures (Section 4.1.2) and measurement instrumentation (Section 4.2.2). Figure 6.13 shows relevant and representative meta-classes provided for this purpose. Custom extensions are possible by extending the respective meta-classes according to the included examples. The `QoSAnnotationModel` contains defini-

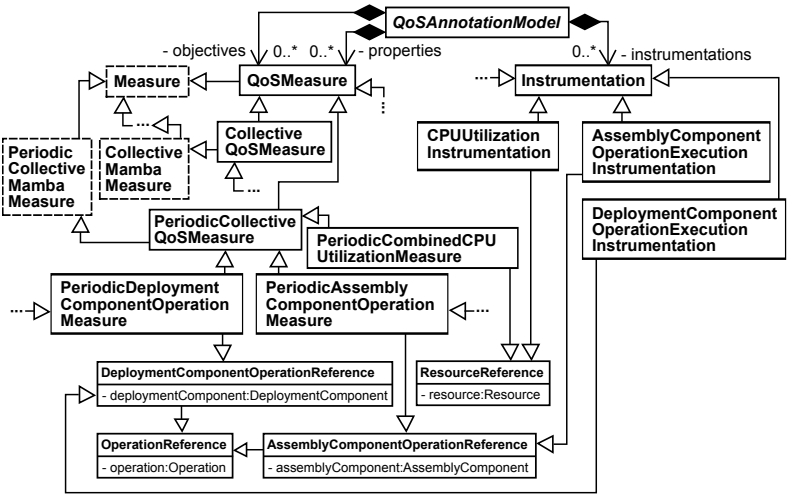


Figure 6.13. Annotations for QoS measures and instrumentation. Meta-classes from the SMM and MAMBA languages are depicted with dashed borders. Helper meta-classes are depicted in smaller size.

tions of QoS objectives (e. g., SLAs) and properties (both elements of type `QoSMeasure`), as well as instrumentation (`Instrumentation`).

For modeling QoS measures, the meta-model agnostic modeling language SMM including our MAMBA extensions (Section 4.1.4) is integrated. The integration is conducted by the meta-class `QoSMeasure` extending the SMM meta-class `Measure`. Note that this already allows to annotate SLAstic models by `QoSMeasures` exploiting all SMM/MAMBA concepts. For convenience, the SLAstic meta-model includes a set of specific meta-classes extending `QoSMeasure` and more specific `Measure` classes. Examples, depicted in Figure 6.13, include QoS measures for operations on assembly- and deployment-level (e. g., invocation counts and response times), as well as CPU utilization. Note that constraints are needed for the concrete `QoSMeasure` meta-classes in order to consistently set the SMM's `Measure` properties, such as the `Scope` and it's `recognizer` in form of an `Operation` (cf. Section 4.1.4).

Meta-classes extending `Instrumentation` express different types of measurement instrumentation for architectural entities contained in SLAstic

models. Figure 6.13 includes three example meta-classes relevant to this thesis. They allow to specify the instrumentation of operations (assembly- and deployment) in order to capture information about their executions, as well as the instrumentation of CPUs to measure their utilization. The resulting measurements for these types of instrumentation are the monitoring events described in Section 6.3.1. Note that usually the instrumentation serves to provide `Measurements` for the `QoSMeasures` (cf. Chapter 9).

Chapter 7

Kieker Framework

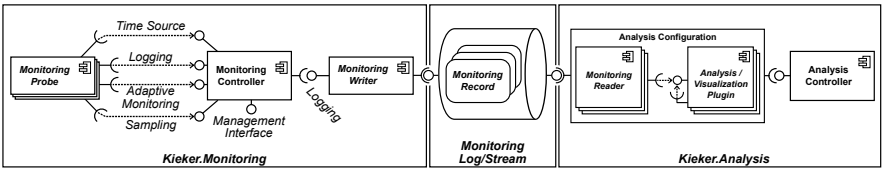
This chapter describes the Kieker framework for application performance monitoring and dynamic analysis of software systems. During the course of this thesis, Kieker evolved from a small tool for monitoring response times of Java software operations into an extensible framework for analyzing the runtime behavior of concurrent and distributed software systems, not being limited to the application in the context of the SLAstic approach.

Note that this chapter focuses on the core concepts of the Kieker framework, including a brief description of its implementation. Additional details about Kieker are available at various places (e. g., [Rohr et al., 2008; van Hoorn et al., 2009c, 2012; Kieker Project, 2014a,b]). This chapter also includes revised contents from these sources, e. g., w.r.t. minor changes in naming and architectural decisions. Chapter 15 reviews Kieker's history, development, and impact.

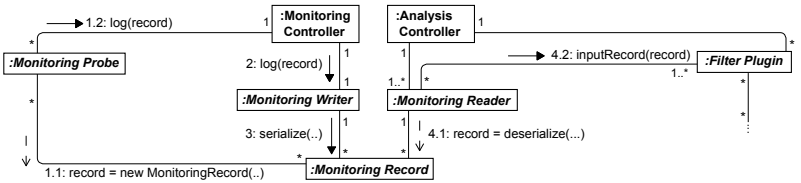
This chapter is structured as follows. Section 7.1 provides an overview of the framework architecture. Section 7.2 details the use of Kieker for logging and analyzing distributed control flow traces. Section 7.3 briefly presents the framework's reference implementation.

7.1 Overview of Framework Architecture

The Kieker framework is structured into a monitoring part, referred to as *Kieker.Monitoring*, and an analysis part, referred to as *Kieker.Analysis*. *Kieker.Monitoring* provides an infrastructure for obtaining and logging measurements from software systems, while *Kieker.Analysis* provides a corresponding infrastructure to analyze these measurements. Figure 7.1a depicts the core components and their assembly, and summarizes the basic vocabulary needed to understand the framework. The Kieker framework is



(a) Kieker's core components and assembly. Extensible components include labels in *italic* font style.



(b) Communication among Kieker framework components for creating, serializing, deserializing, and analyzing a *Monitoring Record*

Figure 7.1. Kieker's core components, assembly, and interactions

designed for extensibility by allowing to use custom implementations of all mentioned components—except for the two controllers.

On the monitoring side, *Monitoring Probes* collect measurements represented as *Monitoring Records*, which a *Monitoring Writer* serializes to a configured *Monitoring Log or Stream*. On the analysis side, *Monitoring Readers* deserialize *Monitoring Records* of interest from the *Monitoring Log/Stream* and pass them to a configurable architecture of *Analysis Plugins*. Both the monitoring and analysis part include dedicated controller components, named *Monitoring Controller* and *Analysis Controller*. Figure 7.1b depicts the core interaction pattern in the Kieker framework for creating, serializing, deserializing, and analyzing a *Monitoring Record*.

The following Sections 7.1.1 to 7.1.3 detail *Monitoring Records* and *Monitoring Logs/Streams*, as well as Kieker's monitoring and analysis components that are part of *Kieker.Monitoring* and *Kieker.Analysis*.

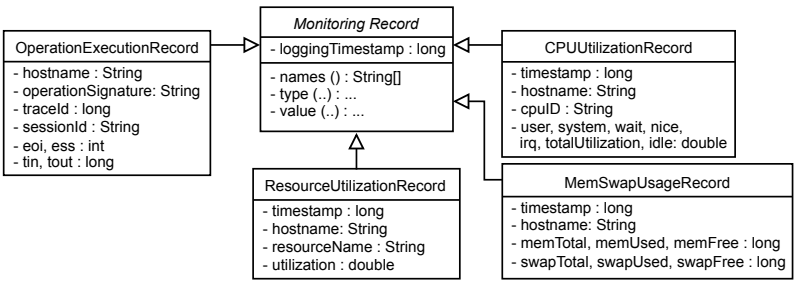


Figure 7.2. Abstract and example *Monitoring Record* meta-classes

7.1.1 Monitoring Records and Monitoring Log/Stream

A *Monitoring Record* is a data structure used to represent data obtained from a single measurement. Informally, a *Monitoring Record* includes a set of named and typed attributes, as well as operations to serialize and deserialize the values associated with the attributes. *Monitoring Records* are typed, i. e., the contained set of attributes is determined by a respective *Monitoring Record* type. Figure 7.2 shows the meta-model, including an abstract meta-class and example concrete meta-classes for *Monitoring Records*. Note that we use a slightly sloppy way of modeling the *Monitoring Record*'s attributes: they are not modeled as references to a dedicated meta-class for attributes but meta-class attributes are used directly. Moreover, we do not distinguish type and instance meta-classes for *Monitoring Records*; types are defined by extending the abstract meta-class, from which instances can be created. The abstract meta-class comprises a single attribute to store the point in time a *Monitoring Record* has been logged (detailed in Section 7.1.2). A *Monitoring Record* includes three operations that serve to access the names, types, and values of the attributes as indicated in Figure 7.2. Figure 7.2 includes four concrete *Monitoring Records*—three for storing measurements of resource utilization, and another one for capturing information on operation executions, including control flow information. For these *Monitoring Records*, corresponding architecture-level monitoring events were introduced in Section 6.3.1 already. However, note that monitoring records only use plain data types instead of referencing to instances of other meta-classes. For example, *OperationExecutionRecords* include the host name and operation

```
ResourceUtilizationRecord;..7730;..7730;SRV0;CPU-0;0.344
ResourceUtilizationRecord;..7740;..7740;SRV0;CPU-0;0.321
ResourceUtilizationRecord;..7750;..7750;SRV0;CPU-0;0.121
OperationExecutionRecord;..7752;SRV1;Catalog.getBook();324;AJKJ;3;2;..7751;..7752
ResourceUtilizationRecord;..7760;..7760;SRV0;CPU-0;0.114
```

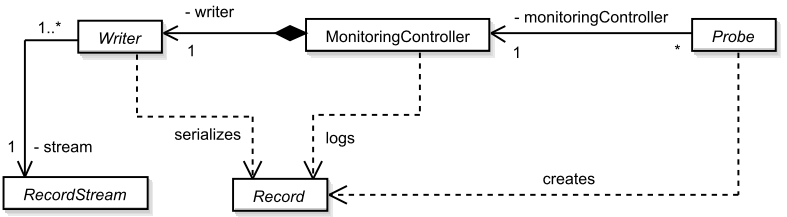
Figure 7.3. Example file system representation of *Monitoring Records*

signature as a string, while the architecture-level events include references to respective entities in the SLAstic model.

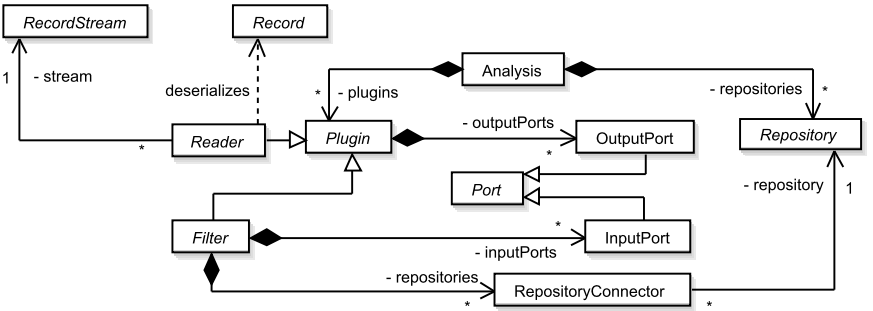
As detailed in Sections 7.1.2 and 7.1.3, *Monitoring Records* are transferred from the monitoring part to the analysis part via a so-called *Monitoring Log or Stream*. A core characteristic of the Kieker approach is that we make no specific assumption about what medium is used for realizing a *Monitoring Log or Stream*. Examples include files in local or distributed file systems, databases, queues/channels provided by message-oriented middleware (MOM), or direct thread communication. The only requirement is that a corresponding pair of *Monitoring Writer* and *Monitoring Reader* exists that allow to serialize and deserialize *Monitoring Record*s to a respective medium-specific representation. Figure 7.3 provides an example file system representation of a *Monitoring Log/Stream*, including *Monitoring Records* conforming to the types included in Figure 7.2. In this case, each row includes the *Monitoring Record* types as well as the list of attribute values— starting with the logging timestamp— in a comma-separated values (CSV) format (using a semicolon as separator).

7.1.2 Monitoring Part

The core components in the monitoring part, called *Kieker.Monitoring*, are *Monitoring Probes*, *Monitoring Controllers*, and a *Monitoring Writer* (Figures 7.1a and 7.4a). A *Monitoring Controller* initializes and controls an instance of a *Kieker.Monitoring* component deployed to a software system. Note that a set of *Monitoring Controllers* is typically used in distributed monitoring scenarios. As depicted in Figure 7.4a, a *Monitoring Controller* is used by a set of *Monitoring Probes* and it has an associated *Monitoring Writer*. A mentioned before (see also Figure 7.1b), the core pattern is that *Monitoring*



(a) Core *Kieker.Monitoring* entities



(b) Core *Kieker.Analysis* entities (omitting the *Analysis Controller*)

Figure 7.4. Core entities of the Kieker monitoring and analysis framework

Probes create *Monitoring Records* and pass them to the *Monitoring Controller*; the *Monitoring Controller* delegates the record to the configured *Monitoring Writer*, which communicates with the respective *Monitoring Log/Stream*.

A *Monitoring Controller* provides a set of operations to *Monitoring Probes*. Examples—grouped into interfaces—are provided in Figure 7.1a, including operations for *a*) getting access to a time source, e. g., to obtain the current time, *b*) passing created *Monitoring Records* to the *Monitoring Writer*, *c*) and maintaining the configuration of sampling-based measurements.

A *Monitoring Probe* implements the measurement logic that collects and possibly preprocesses measurement data from the monitored software system. A *Monitoring Probe* creates *Monitoring Records* and passes these to the associated *Monitoring Controller* using the aforementioned interface. With respect to the foundations on performance measurement described in

Section 4.2, we make no further assumptions about the trigger mechanism (even-driven vs. sampling-based), the instrumentation technique (e. g., direct/indirect code modification), or how the measurement data of interest is obtained (e. g., control flow information or resource utilization using dedicated tools or libraries).

Regardless of their implementation, *Monitoring Writers* implement an interface that is used by the *Monitoring Controller* to delegate the records to be passed to the *Monitoring Log or Stream*. At the time of delegating a *Monitoring Record* to the *Monitoring Writer*, the *Monitoring Controller* sets the *Monitoring Record*'s logging timestamp, mentioned in Section 7.1.1 (see also Figure 7.2).

7.1.3 Analysis Part

The core components in the analysis part, called *Kieker.Analysis*, are *Monitoring Readers*, an *Analysis Controller*, and an *Analysis Configuration* of *Analysis Plugins* (Figures 7.1a and 7.4b). An *Analysis Controller* initializes and controls an instance of the *Kieker.Analysis* component, which serves to analyze the *Monitoring Records* provided by one or more *Monitoring Logs or Streams*. As depicted in Figures 7.1b and 7.4b, *Monitoring Readers* deserialize *Monitoring Records* from a *Monitoring Log or Stream*. A *Monitoring Reader* passes *Monitoring Records* to *Analysis Plugins*, which process the data, e. g., to pass them to other *Analysis Plugins* or to provide appropriate outputs in form of visualizations or alerts. Note that *Monitoring Readers* can be seen as a special kind of *Analysis Plugins*. In addition to *Analysis Plugins*, an *Analysis Configuration* may include *Repositories* (Figure 7.4b), which are associated with *Analysis Plugins* and may be used for sharing state between *Analysis Plugins*, e. g., by connecting to DBMSs or MOM.

We make no specific assumptions about the architectural style used for the *Analysis Configuration* of *Analysis Plugins*, e. g., whether it is event-based, based on pipes and filters, or uses a hybrid style. Figure 7.4b includes the concept for an *Analysis Configuration* based on pipes and filters, as this has been Kieker's focus so far. In this case, *Analysis Plugins* have zero or more *Output Ports*, via which analysis results in form of events may be passed to other *Analysis Plugins*. Note that the type of events in the *Analysis Configuration* is not limited to *Monitoring Records*, even though *Monitoring Readers*—special *Analysis Plugins*—are expected to output *Monitoring Records*

via their *Output Ports*. *Analysis Plugins* that additionally have zero or more *Input Ports* are called *Filters*. *Filters* can be connected to the aforementioned *Repositories*.

7.2 Control Flow Tracing and Analysis

This section describes how the Kieker framework can be used to monitor and analyze distributed control flow traces including timing information. The extracted information can be used to derive models and visualizations about a software system's architecture, usage profile, and performance properties, which may serve as a basis for further analysis.

Note that the description in this section is based on a special type of *Monitoring Records*, namely *OperationExecutionRecord* (see Figure 7.2 and Section 7.2.2), which include information on executed software operations. The approach can also handle other control flow events [Knoche et al., 2012], e. g., operation calls, entries, and exits.

The remainder of this section describes Kieker's meta-model for representing architectural models (Section 7.2.1), details the automatic trace-based model reconstruction (Section 7.2.2), and provides example architectural views generated from the reconstructed models (Section 7.2.3).

7.2.1 System and Trace Meta-Model

Figure 7.5 (page 114) depicts the core parts of the meta-model used inside the *Kieker.Analysis* component to represent the architectural structure of a monitored software system in terms of components and their deployment. Note that the modeling abstractions are quite close to the ones described in Section 6.2. On the type level, a *component type* has a fully-qualified name and implements a set of *operations*. In the simplest case, a component type corresponds to an implementation-level module of the software system, e. g., a class in an object-oriented system. Each operation has a *signature* with a name, a list of modifiers, a list of parameter types, and a return type. *Assembly components* are logical components, having a name and conforming to a component type. Multiple assembly components conforming to the same component type may exist. Each assembly component may be instantiated multiple times as so-called *allocation components*. An allocation

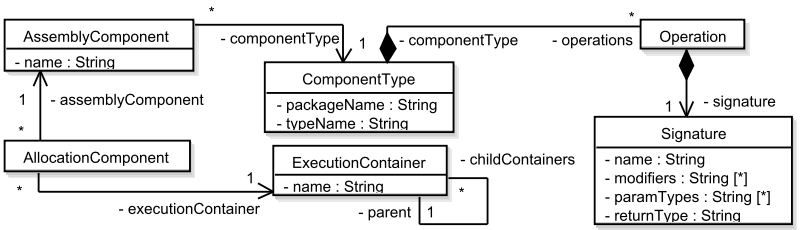


Figure 7.5. Core classes of the meta-model for representing component-based software systems, which is used by Kieker's trace analysis. Note that only the core attributes are included in this diagram.

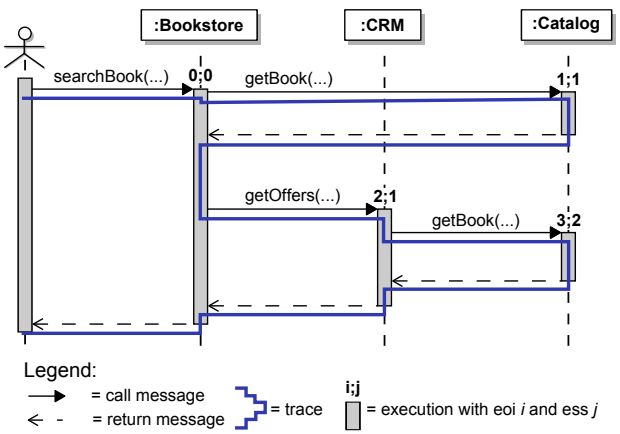


Figure 7.6. Tracing-related terminology in Kieker [van Hoorn et al., 2009c]

component is a deployed instance of an assembly component, located on a so-called *execution container*. An execution container may, e. g., correspond to a physical or virtual machine. It is possible to describe nested execution containers.

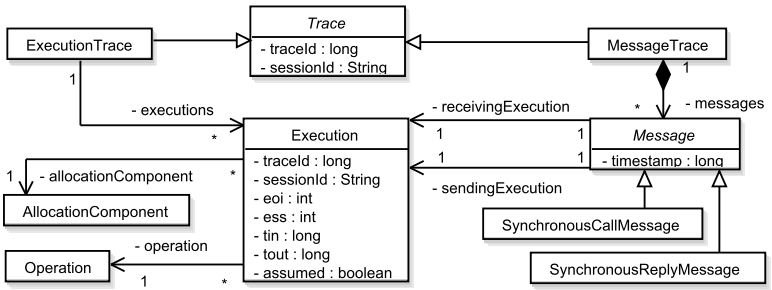


Figure 7.7. Meta-model used by Kieker for representing reconstructed traces.

An *execution* denotes the execution of an operation of an associated allocation component at runtime. The UML sequence diagram in Figure 7.6 includes four executions of three different operations (`getBook()` is called twice). A request to a system-provided operation results in a nested control flow of corresponding executions, referred to as a *trace*. Each execution can be described by a corresponding *call message*, representing an operation call which starts the execution, and a *return message*, representing the end of an execution returning the control flow to the calling execution, as illustrated in Figure 7.6. In *Kieker.Analysis*, two equivalent representations of traces are used internally: *execution traces* and *message traces*. An *execution trace* representation of a *trace* is simply the ordered (by *execution order index* values) sequence of executions (see Figure 7.7). A *message trace* describes a *trace* in terms of an ordered sequence of messages instead of executions. Figure 7.7 shows the relations among *executions*, *messages*, *execution traces*, and *message traces*. The notion of the `eoi` and `ess` values contained in Figures 7.6 and 7.7 will be detailed in the following section.

7.2.2 Logging and Reconstructing Trace Information

Kieker includes the *Monitoring Record* type *OperationExecutionRecord* which can be used to write execution information into the *Monitoring Log/Stream*. As shown in Figure 7.2, an *OperationExecutionRecord* contains information about the executed operation, the host name on which the execution was performed, as well as timestamps, typically with nanosecond resolution, for

the start (`tin`) and end (`tout`) of an execution. Additionally, an *OperationExecutionRecord* includes trace and session identifiers, as well as additional control flow information (`eoi` and `ess`), as detailed below. An example CSV-based serialization of an *OperationExecutionRecord* is included in Figure 7.3. Note that a very basic operation signature is shown in the example, e. g., omitting operation parameters and modifiers.

Kieker includes different *Monitoring Probe* types for logging *OperationExecutionRecords* within an application and provides efficient facilities for attaching a unique trace identifier to the thread executing a service request, which is then contained in any *OperationExecutionRecord* of that trace (see Figure 7.7).

If the reconstruction of traces from a *Monitoring Log/Stream* containing *OperationExecutionRecords* would only include the trace information presented so far, we would require the following assumptions: *a*) no two execution start or end time events (`tin`/`tout`) within the same trace occur at the same time; and *b*) clocks in a distributed system are perfectly synchronized (both with respect to the respective time resolution). Since both assumptions cannot be guaranteed in realistic environments, Kieker includes efficient facilities to attach two additional parameters to any *OperationExecutionRecord* in order to log the information needed to reconstruct (distributed) traces from the *Monitoring Log/Stream* reliably: an execution order index *(eoi)* and an execution stack size *(ess)*.

1. An execution with an *eoi* value *i* denotes the *i*-th *execution* started within a *trace* (starting with the value 0).
2. An execution with an *ess* value *j* denotes an execution that was started when the depth of the calling stack for the corresponding *trace* was *j*.

The executions shown in the example trace in Figure 7.6 are annotated with the corresponding *eoi* and *ess* values. Note that while an *execution order index* is unique within a *trace*, an *execution stack size* value can—and usually does—occur more than once.

While it is straightforward to derive *execution traces* from the *Monitoring Log*, for later analysis it is usually easier to derive analysis models from *message traces*. In *Kieker.Analysis*, *execution traces* are derived from the *Monitoring Log/Stream* and then transformed into equivalent *message trace* representations from which analysis models and diagrams are created.

7.2.3 Reconstructing Architectural Views

From the obtained system models and traces (instances of the meta-model in Figures 7.5 and 7.7), structural and behavioral architectural views can be created. As representative examples, this section describes sequence diagrams and calling dependency graphs. Example visualizations, created by the Kieker implementation described in Section 7.3, are shown in Figure 7.8. Additional visualizations derived in a similar way, e. g., dynamic call trees and Markov chains, are also described in our previous works [Rohr et al., 2008; van Hoorn et al., 2009c].

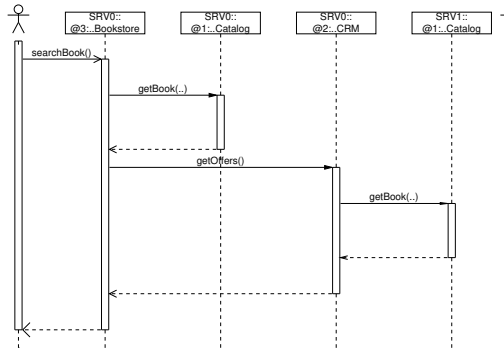
Sequence Diagrams

UML sequence diagrams provide a dynamic architectural viewpoint in terms of interactions among runtime objects implementing software services. In Figure 7.6 of the previous section, we used a sequence diagram to illustrate the trace-related terminology needed to define *execution traces* and *message traces*. Message traces can be transformed to UML sequence diagrams in a straightforward way. Figures 7.8a and 7.8b show UML sequence diagrams generated by a Kieker visualization component for the running Bookstore example. Given the timing information included in the execution traces, the sequence diagrams could easily be augmented with additional data, e. g., observed response times. The UML specification [Object Management Group, Inc., 2013b] and the UML profiles for performance [Object Management Group, Inc., 2005, 2011c] suggest appropriate notations for performance annotations.

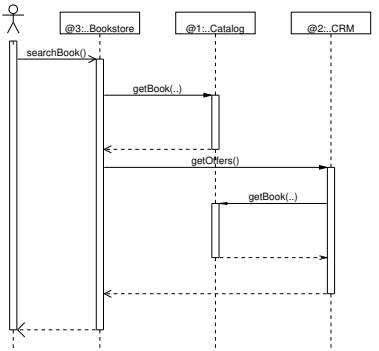
When analyzing traces from the *Monitoring Log or Stream*, a valuable initial analysis is to determine the *trace equivalence classes*. Informally, a trace equivalence class contains all traces that are equal in terms of the control flow structure, i. e., the sequence diagrams of all *traces* in an equivalence class are identical. Based on the message traces this analysis can be implemented efficiently.

Dependency Graphs

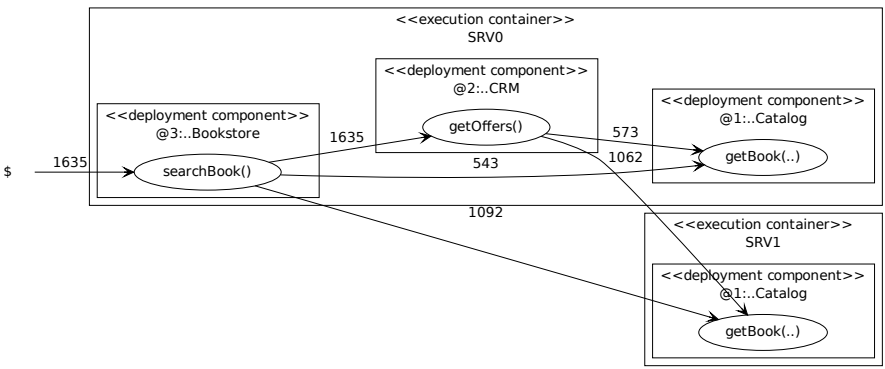
While sequence diagrams, or execution traces, provide a view on the *sequence* of interactions among objects in a single trace (or scenario/use case), it is often desirable to analyze this information in an aggregated form.



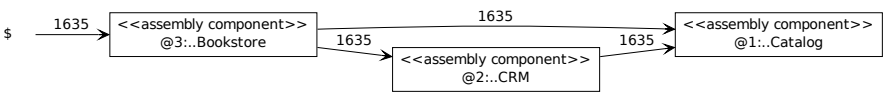
(a) *Deployment-level* sequence diagram



(b) *Assembly-level* sequence diagram



(c) *Deployment-level* operation dependency graph



(d) *Assembly-level* component dependency graph

Figure 7.8. Selected visualizations generated by Kieker based on reconstructed trace information. For a complete overview of visualizations supported by Kieker, please refer to the Kieker user guide [Kieker Project, 2014a].

Interactions among objects constitute runtime dependencies among these system entities, which can be described using weighted directed dependency graphs: each entity is assigned a node and each dependency relation an edge; the edge is directed from an entity using a particular service to the entity providing that service; the edges are augmented with the total number of call actions among the respective entities observed in the considered set of traces.

We implemented a *Kieker.Analysis* component which computes dependency graphs from a set of message traces. These dependency graphs are then available for further analysis or visualization. Figures 7.8c and 7.8d show dependency graphs generated by Kieker, visualizing calling dependencies among operations (deployment-level) and components (respectively) of the Bookstore application. The diagrams provide aggregated views of the runtime dependencies observed in 1635 traces.

7.3 Framework Implementation

This section briefly presents the implementation of the framework described in Section 7.1. The implementation of the framework, mainly based on Java, is publicly available as open-source software [Kieker Project, 2014b]. The description in this section is based on Kieker 1.9, which is the most recent version at the time of writing. Note that a far more comprehensive description of the implementation and its usage is available, e. g., as part of the user guide [Kieker Project, 2014a].

The Kieker implementation can be divided into two parts: first, the framework conforming to the description in Section 7.1, which provides extension points for custom extensions to be used with the framework; second, concrete components that have been developed for specific purposes by making use of the framework-provided extension points. Based on the afore-described Figure 7.1, Figure 7.9 depicts Kieker's core components, extension points, and features.

The following sections describe the extensible framework architecture including the extension points (Section 7.3.1), provide an overview about the framework components developed based on the extension points (Section 7.3.2), and briefly summarizes developments for non-Java platforms (Section 7.3.3).

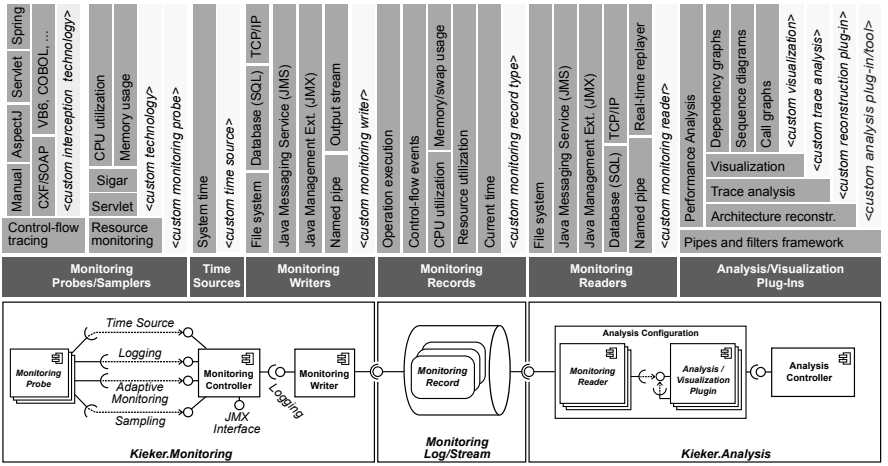


Figure 7.9. Kieker's core components, extension points, and features (based on [Kieker Project, 2014a])

7.3.1 Extensible Framework Architecture

This section describes the extensible framework architecture comprising non-replaceable monitoring and analysis controllers as well as extension points for custom components.

Non-Replaceable Monitoring and Analysis Controllers

The *Monitoring Controller* provides interfaces for *a*) logging *Monitoring Records* using the configured *Monitoring Writer*, *b*) retrieving the current time via the configured *Time Source*, *c*) scheduling and removing periodic samplers, *d*) activating and deactivating *Monitoring Probes* at runtime, as well as *e*) enabling, disabling, and terminating monitoring. The *Monitoring Controller* is configured via well-documented configuration properties that are evaluated during initialization. Multiple *Monitoring Controllers* can be used in parallel, even within the same Java Virtual Machine (JVM).

The *Analysis Controller* provides interfaces for *a*) defining *Analysis Configurations* by registering and connecting *Analysis Plugins*, including *Monitoring*

Readers, *Filters*, and *Repositories*, *b*) running and terminating executions of the *Analysis Configuration*, *c*) as well as for persisting and loading *Analysis Configurations*.

Extension Points

Extension points can be divided into those for custom components in the *Kieker.Monitoring* and *Kieker.Analysis* part, as well as the *Monitoring Records* used in both parts.

- *Monitoring Records.* For the definition and use of *Monitoring Record* types, Kieker provides an interface (`IMonitoringRecord`), which—according to Section 7.1.1—declares operations for getting and setting a logging timestamp as well as to serialize and deserialize a *Monitoring Record*. An abstract class (`AbstractMonitoringRecord`) implementing this interface is provided by Kieker, which already implements the management of the logging timestamp and provides some convenience functionality for extending classes. Different variants for the implementation of custom *Monitoring Record* types exist, e. g., to support the implementation of immutable classes.

- *Monitoring*

  Extension points in *Kieker.Monitoring* are provided for custom *Monitoring Probes*, *Monitoring Writers*, and *Time Sources*. Dedicated interfaces (`IMonitoringProbe`, `ISampler`) and abstract classes (`AbstractMonitoringWriter`, `AbstractTimeSource`) need to be implemented or extended respectively. Custom *Monitoring Probes* (including samplers) are typically technology-specific w.r.t. the techniques used for instrumentation and for gathering the runtime data. The common part is the creation of *Monitoring Records* and their delivery to the *Monitoring Controller*. Convenience functionality for the implementation of asynchronous *Monitoring Writers* is provided.

- *Analysis*

  Extension points in *Kieker.Analysis* are provided for custom *Monitoring Readers*, *Filters*, and *Repositories*. Dedicated abstract classes (`AbstractReaderPlugin`, `AbstractFilterPlugin`, and `AbstractRepository`) already provide common functionality and need to be extended. In each case,

methods for initialization and termination need to be implemented. For *Monitoring Readers*, it is additionally required to implement the functionality to deserialize *Monitoring Records* from the respective *Monitoring Log or Stream*. *Monitoring Records* are then provided via a declared *Output Port*. For *Filters*, the functionality to be triggered by the events received via the *Input Ports* needs to be implemented, including the delivery of events via *Output Ports*. Kieker's pipes-and-filters API makes use of Java annotations to declare an *Analysis Plugin*'s configuration properties including default values, *Input Ports* and *Output Ports*, etc. Declarations of *Input Ports* and *Output Ports* include definitions of a port name, description, and a list of accepted event types. Note that the event types are not limited to *Monitoring Records* but may be any Java type.

7.3.2 Framework Components

Building on the previously described extension points, a number of components have been developed. This section aims to provide an overall overview about the available set of components that are contained in the Kieker distribution.

Monitoring Records

The most important *Monitoring Record* types provided by Kieker are those that can be used for representing control flow events and resource utilization, including the ones from Figure 7.2.

Monitoring

- Particularly two types of *Monitoring Probes* are currently included in the Kieker distribution. *Monitoring Probes* of the first type focus on tracing of events observed in application-internal control flows (Section 7.2); *Monitoring Probes* of the second type focus on monitoring the utilization of system-level resources. With respect to control flow tracing, the included *Monitoring Probes* make use of AOP-based instrumentation and interception techniques employing technologies such as AspectJ [The Eclipse Foundation, 2014], Spring interceptors [SpringSource, 2014], Servlet filters [Oracle, 2014a], as well as the interceptors provided by

the Apache CXF [The Apache Foundation, 2014] library for SOAP-based web services. These technologies are in wide-spread use in Java-based enterprise application systems (Section 3.3). *Monitoring Probes* based on the different technologies can be used in combination, which is the case in the evaluation Chapters 12 and 13. The CXF-based *Monitoring Probes* allow distributed tracing across remote machines. *Monitoring Probes* for measurements of resource utilization exist for CPUs and memory. These probes make use of the sampling interface provided by the *Monitoring Controller*, i. e., they are triggered in configurable intervals. Respective Servlet filters exist that ease the registration of the *Monitoring Probes* in Servlet containers.

- A number of *Monitoring Writers* are included in the distribution, which can be used for *Monitoring Logs or Streams* realized employing local or distributed file systems, SQL databases, JMX, JMS, direct memory communication, and via TCP/IP-based network connections. For both the file system and the database *Monitoring Writers* synchronous and asynchronous implementations exists. Different variants of the file system *Monitoring Writers* exist, which write CSV-based, binary, or compressed representations. File system *Monitoring Writers* are particularly useful for offline analysis, where *Monitoring Records* are first collected and processed later in a batch-like mode. On the other hand, the *Monitoring Writers* based on Java Management Extensions (JMX), Java Message Service (JMS), TCP/IP, and direct memory communication are most useful for online analysis, when the *Monitoring Records* are analyzed while the system is running and producing additional *Monitoring Records*—including settings with monitoring and analysis being deployed on separate machines.

- *Time Sources* based on the current system time are available, which use the milliseconds and nanoseconds included in the Java Runtime Environment (JRE). Examples for other useful *Time Sources*, which are currently not included in the Kieker distribution, obtain the current time from a simulation engine (Section 11.3) or a Network Time Protocol (NTP) server (developed by Zobel [2012]).

Analysis

- ▻ According to the previously mentioned *Monitoring Writers*, corresponding *Monitoring Readers* exist, i. e., for file systems, SQL databases, JMX, JMS, TCP/IP, and direct memory communication. Note that the pipes-and-filters architecture allows the use of multiple *Monitoring Readers*, which allows to merge multiple *Monitoring Logs or Streams*, e. g., from different machines using the file system as a *Monitoring Log*.
- ▻ Various useful *Filters* have been developed ranging from simple *Filters* for counting events, over *Filters* that select events based on specific attributes (e. g., trace identifiers or timestamps), to *Filters* that serve the trace reconstruction and analysis, as described in Section 7.2. With respect to the latter, *Filters* are available that provide the architecture visualization capabilities described in Section 7.2.3, including calling dependency graphs on deployment, component, and operation level, as well as sequence diagrams and call trees. The examples in Figure 7.8 have been created with these *Filters*. The *Repository* supporting the trace analysis maintains an instance of the system model as described in Section 7.2.1. A *Filter* for delaying the incoming stream of *Monitoring Records* according to the time resolution from the original *Monitoring Log or Stream* is included as well.

7.3.3 Monitoring Adapters for Other Platforms

In order to support the monitoring of platforms other than Java, respective adapters have been developed, including C# [Magedanz, 2011], Visual Basic 6 (VB6) [van Hoorn et al., 2011b], and COBOL [Richter, 2012; Knoche et al., 2012]. The developed *Monitoring Probes* focus on tracing of application-internal control flows and produce *Monitoring Records* according to those provided for Java systems. The adapters either use the Java-based Kieker framework directly via a language-specific bridge to Java or write an intermediate representation of the monitoring data, which is transformed to the respective *Monitoring Records*. Similar to this, adapters for different monitoring facilities have been developed, e. g., RRDtool as detailed in Section 12.3.

Chapter 8

SLAstic Framework

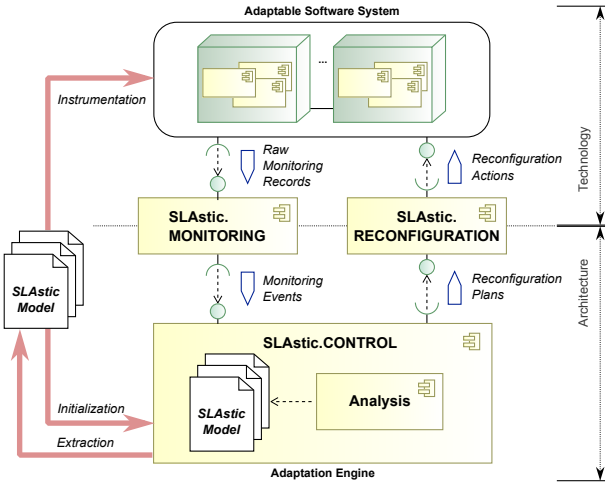
This chapter describes the SLAstic framework, which implements a MAPE-K-based (Section 3.4.1) self-adaptation platform tailored to the model-driven and architecture-based online capacity management approach developed in this thesis.

The chapter is structured as follows. Section 8.1 provides an overview of the framework architecture. Sections 8.2 to 8.5 describe the core framework components: Model Manager, Monitoring Manager, Adaptation Controller, and Reconfiguration Manager. Section 8.6 briefly presents the framework's proof-of-concept implementation.

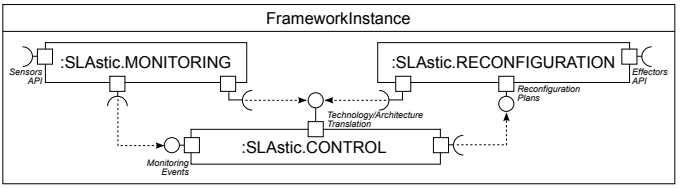
8.1 Overview of Framework Architecture

The framework is composed of the following three main processing components: Monitoring Manager (also denoted as SLAstic.Monitoring), Adaptation Controller (SLAstic.Control), and Reconfiguration Manager (SLAstic.Reconfiguration). Figure 8.1 provides an illustrative (Figure 8.1a) and a more technical (Figure 8.1b) view on the framework's top-level architecture, including the three mentioned components, their relationships, and their external integration with the adaptable software system. Figure 8.1a includes related activities for model-driven instrumentation and model extraction, which will be detailed in Chapter 9.

A software system to be controlled by the SLAstic framework needs to be equipped with two types of interfaces—one for providing monitoring events based on a customized monitoring instrumentation and a second for receiving reconfiguration triggers. Via these interfaces, the system interacts with the framework components SLAstic.Monitoring and SLAstic.Reconfiguration, which map to the *M* and *E* activities as part of the MAPE-K



(a) Components and context



(b) Composite structure

Figure 8.1. Top-level views on the SLAstic framework architecture

control loop. Figure 8.1a depicts a separation between two different layers of abstraction: the technology-specific implementation of the controlled system[1] vs. its architectural representation and reasoning. This separation is a core property of our approach. We make no specific assumptions about the technologies that are used for the implementation of the controlled system and its interfaces. The SLAstic.Control component, which maps to MAPE-K's *A* and *P* activities, solely reasons based on architectural runtime models of the controlled system. A Model Manager, which is part of

[1]We use the terms *controlled software system* and *adaptable software system* interchangeably.

SLAstic.Control, provides DBMS-like access to these runtime models that conform to the meta-model introduced in Chapter 6 (*K* in MAPE-K). The components SLAstic.Monitoring and SLAstic.Reconfiguration translate between the two abstraction levels based on a bidirectional mapping between architectural and implementation entities, which is also maintained by the Model Manager.

Having a first idea about the framework architecture and integration, the supported workflow is as follows. The Monitoring Manager continuously receives technology-specific *Monitoring Records*—e. g., Kieker monitoring records (Section 7.1.1)—from the controlled software system and translates them into architecture-level SLAstic monitoring events described in Section 6.3. Based on these events and the architectural knowledge captured in the runtime models, a configuration of analysis components contained in the Adaptation Controller continuously performs model-based performance and workload analyses, performance predictions, and adaptation planning. If the adaptation planning decides on an adaptation to be executed, it creates a reconfiguration plan conforming to the respective meta-model described in Chapter 6 and passes this plan to the Reconfiguration Manager. The Reconfiguration Manager is responsible for the transactional execution of this plan, which includes the translation of architecture-level reconfiguration actions into calls to the respective technology-specific interfaces of the adaptable software system.

Starting with the Model Manager, the following Sections 8.2 to 8.5 will detail the core framework components. Based on this, we developed the Java-based proof-of-concept implementation presented in Section 8.6.

8.2 Model Manager

The Model Manager provides DBMS-like functionality for the SLAstic runtime model and the bidirectional mapping between architecture and implementation. The functionality includes meta-model-specific services like CRUD (create, read, update, and delete) operations and constraint validation on SLAstic models, as well as technical services like persistence and transaction management (cf. Section 2.3 on model repositories).

As depicted in Figure 8.2, the Model Manager is decomposed into nine subcomponents, detailed in the remainder of this section. A dedicated

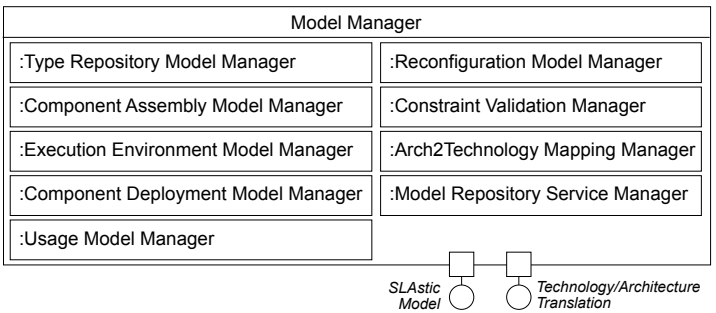


Figure 8.2. Decomposition of the Model Manager into subcomponents

Model Manager exists for the SLAstic system and usage models. The five components on the left-hand side of the diagram—Type Repository Model Manager, Component Assembly Model Manager, Execution Environment Model Manager, Component Deployment Model Manager, and Usage Model Manager—include the CRUD functionality for the corresponding submodels of the SLAstic meta-model. These submodel managers have access to each other, in order to satisfy the inter-model dependencies.

8.2.1 Type Repository Model Manager

The Type Repository Model Manager manages an instance of the SLAstic meta-model's type repository (Section 6.2.1), including the following entities: Component Types, Connector Types, Interfaces, Execution Container Types, Network Link Types, and Resource Types. It provides operations to create, register, and lookup these entities and their properties.

8.2.2 Component Assembly Model Manager

The Component Assembly Model Manager manages an instance of the SLAstic meta-model's Component Assembly Model (Section 6.2.2), including the following entities: Assembly Components, Assembly Connectors, (as well as System-Provided and System-Required) Interfaces. Like the Type Repository Model Manager, the Component Assembly Model Manager provides corresponding create, register, and lookup operations. Moreover,

it provides operations to connect an Assembly Component with another Assembly Component, with a System-Provided Interface, or with a System-Required Interface using an appropriate Assembly Connector.

8.2.3 Execution Environment Model Manager

The Execution Environment Model Manager manages an instance of the SLAstic meta-model's Execution Environment Model (Section 6.2.3), including the following entities: Execution Containers (including contained Resources) and Network Links. In addition to the usual create, register, and lookup operations, the Execution Environment Model Manager provides add and remove operations for the list of allocated Execution Containers (*allocatedExecutionContainers*, see Figure 6.6), and operations to connect Execution Containers via Network Links.

8.2.4 Component Deployment Model Manager

The Component Deployment Model Manager manages an instance of the SLAstic meta-model's Component Deployment Model (Section 6.2.4), including Deployment Components. In addition to the usual create, register, and different variants of lookup operations, the Component Deployment Model Manager provides operations to delete and to migrate a Deployment Component. The delete operation does not really remove a Deployment Component from the model but marks it inactive.

8.2.5 Usage Model Manager

The Usage Model Manager manages an instance of the SLAstic meta-model's Usage Model (Section 6.3.3), including information on calling relationships and calling frequencies. It provides operations to lookup information contained in the model by passing the decorated model entities (e. g., Connector, Operation, Interface, Signature), as well as to increment frequencies (with implicit creation of model entities).

8.2.6 Constraint Validation Manager

The Constraint Validation Manager implements the functionality to check whether the SLAstic meta-model constraints are satisfied for the current version of the SLAstic runtime model. Also, it is used to test pre- and post-conditions of the previously mentioned model change operations.

8.2.7 Reconfiguration Model Manager

The Reconfiguration Model Manager provides access to an instance of the SLAstic meta-model's Reconfiguration Model (Section 6.4.2)—including capabilities and properties—and translates results of executed reconfiguration operations (Section 6.4.1) on the SLAstic model.

8.2.8 Arch2Technology Mapping Manager

The Arch2Technology Mapping Manager maintains bidirectional mappings between technology and architectural identifiers, separated by SLAstic meta-model entity types, e. g., for execution containers and assembly components. It provides an operation to register a mapping, and two operations to lookup an identifier by providing an architecture-specific identifier or technology-specific identifier respectively.

8.2.9 Model Repository Service Manager

The Model Repository Service Manager connects to a model repository technology and wraps its technology-specific services—described in Section 2.1—in an API. These services are provided by state-of-the-art meta-model-agnostic model repository technologies (Section 2.1) and have been investigated by Kiel [2013] in the context of this thesis (Section 5.3.1).

8.3 Monitoring Manager

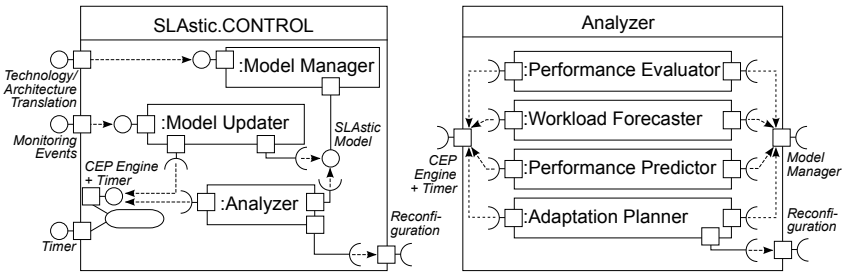
The Monitoring Manager connects to a monitoring infrastructure in order to receive implementation-level monitoring events. These monitoring events are transformed into SLAstic monitoring events (Section 6.3) that

are related to architectural entities from the SLAstic runtime model maintained by the Model Manager. For this purpose, both interfaces of the Model Manager (Figure 8.2) are used in order to get access to the *a*) SLAstic runtime model and *b*) the mappings between identifiers on implementation-level and architecture-level. The created monitoring events are sent to the Adaptation Controller for further processing via a dedicated interface (Figure 8.3a), as detailed in the following Section 8.4.

In this work, we assume that the Monitoring Manager employs Kieker as the monitoring infrastructure. Hence, it receives *Monitoring Records* and executes a transformation function *MonitoringRecord* → *Event* for each record from implementation level (`MonitoringRecord`, Section 7.1.1) to architecture level (`Event`, Section 6.3.1). Relevant to this thesis are the Kieker *Monitoring Records* for resource measurements and operation executions (Section 7.1.1), which are transformed into corresponding SLAstic monitoring events for operation executions and resource usage (Section 6.3). This transformation will be detailed in Chapter 9.

In this thesis, we distinguish two cases with respect to the completeness of the SLAstic runtime model. In the first case, the SLAstic runtime is complete in that each implementation-level entity has a representation in an architectural entity, i. e., for each incoming monitoring event, a corresponding architectural entity exists and can be looked up in the Model Manager. This case is enforced by the model-driven instrumentation and analysis support, described in Chapter 9. In the second case, the SLAstic runtime model is not complete—in its extremist form, it is even completely empty. Here, it is required to create model entities on-demand, before a monitoring event can be created. This second case will be detailed in Chapter 9.

Kieker's support for replaying recorded *Monitoring Logs* is the basis for operating the SLAstic framework in an offline mode. Note that the Monitoring Manager may be connected to other monitoring facilities, as well. Kieker me be also used to import the data from other monitoring infrastructures, as we will see in this thesis (e. g., RRDtool).



(a) Adaptation Controller **(b)** Analyzer

Figure 8.3. Composite structure of the Adaptation Controller and the Analyzer

8.4 Adaptation Controller

Figure 8.3 depicts the composite structure of the Adaptation Controller. On the top-level (Figure 8.3a), it comprises a Model Manager, a Model Updater, and an Analyzer component, as well as a Complex Event Processing (CEP) Engine and Timer functionality. The Model Manager, as detailed in Section 8.2, provides access to the SLAstic runtime model as an architectural view on the system's structure and usage. The Model Updater first processes the incoming monitoring events to update the runtime model—mainly w.r.t. system usage (Section 6.3). Moreover, it dispatches the monitoring events to the CEP Engine to make them available for the Analyzer component, which—based on these events and the knowledge contained in the runtime model, performs the following tasks: performance evaluation, workload forecasting, performance prediction, and adaptation planning (Figure 8.3b). The following Sections 8.4.1 to 8.4.3 detail the components of the Adaptation Controller.

8.4.1 Complex Event Processing (CEP) Engine and Timer

The Complex Event Processing (CEP) Engine provides functionality to publish and to analyze streams of events [Cugola and Margara, 2012]. For the Adaptation Controller, this provides the platform for asynchronous communication between the components. For this thesis, we assume that an

```
select
  current_timestamp as currentTimestampMillis,
  deploymentComponent.assemblyComponent,
  count(*)
from DeploymentComponentOperationExecution.win:time(10 sec)
group by deploymentComponent.assemblyComponent
output all every 5 seconds
```

Figure 8.4. EPL statement to periodically collect the number of operation executions for each Assembly Component observed within the past 10 seconds every 5 seconds.

event is an object with named properties, extending the meta-class *Event*, i. e., it already includes the monitoring events introduced in Section 6.3. Note that an event's properties are not limited to plain types, but may include references to other objects—e. g., entities from the SLAstic runtime model. An event is published by passing it to a dedicated method provided by the CEP Engine. Components create SQL-like CEP statements and register them with the CEP Engine. Even though we are not making specific assumptions about a CEP technology, we will use the Event Processing Language (EPL), which is part of the Esper CEP component, as a representative language for expressing CEP statements. A detailed EPL documentation is provided by the Esper Team and EsperTech, Inc. [2014]. For each registered CEP statement, a component must provide a handler method, which is called by the CEP Engine passing the result objects.

As an example, Figure 8.4 shows an EPL statement, which serves to periodically (every 5 seconds) obtain the number of operation executions per Assembly Component observed in the past 10 seconds. The example depicts the possibility to navigate the model—in this case to access the Assembly Component via the Deployment Component referenced by the operation execution event (Section 6.3.1).

The Timer is the central component to maintain the current time assumed by the Adaptation Controller. First, it provides a method for the other components to obtain the current time, given as the number of time units elapsed since the UNIX epoch (00:00:00 UTC on January 1, 1970). In case the SLAstic framework operates in online mode, the current time represents the

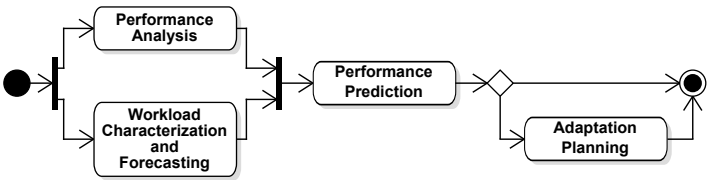


Figure 8.5. Activity diagram of the (proactive) analysis phase. The activities execute in parallel. The structure depicts the data dependencies.

current system time on the computer executing the Adaptation Controller. In offline mode, the current time is determined by an external trigger, which is, for instance, based on the most recent timestamp observed in incoming events by the SLAstic framework. The Timer includes a method to receive time triggers.

8.4.2 Model Updater

The Model Updater receives architecture-level monitoring events from the Monitoring Manager and, based on these events, updates the SLAstic runtime model using the Model Manager and emits existing and new events to the CEP Engine. Of particular concern for the Model Updater ist the update of the Usage Model, including information on calling relationships and calling frequencies.

Note that updates of the runtime model are also performed by the Monitoring Manager. However, while the Monitoring Manager copes with the mapping between technology and architecture, the Model Updater only works with architectural entities and events.

8.4.3 Analyzer

The Analyzer includes components executing the analysis activities focusing on capacity management, namely *performance analysis*, *workload characterization and forecasting*, *performance prediction*, and *adaptation planning*. Figure 8.5 depicts the data flow between the four activities assuming a proactive scenario. As depicted in Figure 8.3b, the Analyzer includes a dedicated

subcomponent for each of the activities. These four subcomponents will be detailed in the following sections.

Performance Evaluator

The Performance Evaluator is responsible for the continuous assessment of the controlled system's performance properties, including current observations and predictions. Particularly, the detection and alerting of violations to the internal and external performance requirements (SLAs), specified as part of the SLAstic meta-model's QoS Model (Section 6.5). In addition to this, the Performance Evaluator logs performance and efficiency statistics for online or offline analysis and visualization.

Workload Forecaster

The Workload Forecaster is responsible for the continuous characterization and forecasting of workload parameters from the controlled software system. This activity includes the analysis of the workload observed in the elapsed period and, based on this, estimating near-future trends. Hence, it implements techniques like the ones described in Section 4.4, e. g., based on time series analysis. Parameters for workload characterization and forecasting are specified as part of the SLAstic meta-model's QoS Model. Note that a Workload Forecaster is particularly useful for proactive capacity planning scenarios in combination with online performance prediction executed by the Performance Predictor described below.

Performance Predictor

The Performance Predictor is responsible for continuous prediction of performance measures of the controlled system for the near future. This activity uses results from the Workload Forecaster and architectural information captured in the SLAstic runtime model as inputs for the predictions. It employs model-based performance prediction techniques like the ones described in Section 4.5. As mentioned before, the Performance Predictor is particularly needed for proactive capacity management.

Adaptation Planner

The Adaptation Planner is responsible for detecting and planning required adaptations of the controlled software system in order to ensure that desired high-level goals are satisfied. Adaptation planning is based on the definition of objectives, e. g., optimizing resource efficiency, as well as strategies and tactics to meet these objectives. All these aspects may, for instance by expressed using the S/T/A modeling language described in Section 3.4.3. For example, the Adaptation Planner may follow a rule-based policy to plan adaptations based on reactive or proactive SLA violation events emitted by the Performance Evaluator. The output of the adaptation planner is a reconfiguration plan that, as described in Section 6.4.1, determines the sequence of reconfiguration actions to be executed. In order to determine suitable reconfiguration plans, the Adaptation Planner typically makes use of similar techniques employed by the aforedescribed components, e. g., model-based software performance prediction.

8.5 Reconfiguration Manager

The Reconfiguration Manager is responsible for executing the reconfiguration plans created by the Adaptation Planner. Therefore, it interprets the reconfiguration plan, which determines the sequence of reconfiguration operations to be executed. As a reconfiguration operation refers to architectural entities from the SLAstic runtime model (Section 6.4), the corresponding implementation-level counterparts need to be looked up using the Model Manager's bidirectional mapping between architecture and implementation. Additional details about the Reconfiguration Manager will be provided in Section 10.2.3.

8.6 Framework Implementation

We developed a Java-based implementation for the SLAstic framework described in the previous sections. Similar to the implementation of the Kieker framework described in Section 7.3, the SLAstic framework implementation includes a basic set of components, being designed for reusability and extensibility. The SLAstic implementation is publicly available as open source

software as part of the Kieker project, e. g., via van Hoorn [2014]. Various framework components developed for the SLAstic framework implementation that served to be useful also for other purposes have been integrated into the Kieker framework.

The remainder of this section is structured as follows. Section 8.6.1 provides an overview of the reusable and extensible framework architecture. Section 8.6.3 focuses on the configuration and setup of framework instances. Section 8.6.4 briefly describes the integration with the Kieker framework. Section 8.6.5 gives examples for concrete framework components that have been implemented. Note that we provide implementation details only to a degree that is needed for this thesis. Technical details, particularly on how to use the framework, are provided as part of the framework documentation and associated examples.

8.6.1 Overview

Matching the framework's conceptual high-level architecture (Figure 8.1), a framework instance is composed of *a*) a Monitoring Manager, *b*) an Adaptation Controller, and *c*) a Reconfiguration Manager. According to Figure 8.1, the Adaptation Controller is decomposed into a Model Manager, a Model Updater, and an Analyzer. The latter is further decomposed into a Performance Evaluator, a Workload Forecaster, a Performance Predictor, and an Adaptation Planner. For each of these components, the framework includes an abstract Java type that declares the respective interface and provides basic functionality. Custom classes must implement these abstract classes. Basic implementations exist for each of the types, which may be used in cases where a component's functionality is not needed, e. g., workload forecasting or performance prediction in reactive adaptation scenarios. Table 8.1 lists the names of the abstract and basic Java types. A super type implemented by each framework component lets components access their configuration properties and their dedicated subdirectory within the results directory created for each framework execution. Moreover, it declares methods for *a*) initialization, *b*) startup, and *c*) termination, which are called by the framework logic.

To summarize, similar to Kieker's implementation described in Section 7.3, the SLAstic framework implementation consists of a generic (core) part that serves the basis for custom extensions as well as concrete frame-

Table 8.1. Implementation classes for conceptual framework components. Note that the name prefixes *I* and *Abstract* for interfaces and abstract classes respectively are omitted for better readability.

Conceptual Component	**Java type** (abstract and default implementation)
Monitoring Manager	*{Abstract|Dummy}MonitoringManagerComponent*
Adaptation Controller	*{Abstract|Basic}ControlComponent*
∟ Model Manager	*{Abstract|Dummy}ModelManager*
∟ Model Updater	*{Abstract|Dummy}ModelUpdaterComponent*
∟ Analyzer	*{Abstract|Basic}AnalysisComponent*
∟ Performance Evaluator	*{Abstract|Dummy}PerformanceEvaluatorComponent*
∟ Workload Forecaster	*{Abstract|Dummy}WorkloadForecasterComponent*
∟ Performance Predictor	*{Abstract|Dummy}PerformancePredictorComponent*
∟ Adaptation Planner	*{Abstract|Dummy}AdaptationPlannerComponent*
Reconfiguration Manager	*{Abstract|Dummy}ReconfigurationManagerComponent*

work components to be reused. For example, the Model Manager already provides a basic integration of the Eclipse Modeling Framework (EMF) technologies, e. g., for loading and storing model instances. A concrete Model Manager has been developed for the SLAstic meta-model described in Chapter 6, using the Java classes generated from the Ecore meta-model. Concrete Model Managers for other meta-models, e. g., PCM, have been developed as well. Also for the Reconfiguration Manager, this extensibility is helpful, as it is partially technology-specific and depends on the controlled system's effector APIs. On the other hand, extensibility does not play a primary role for the Monitoring Manager, as the Kieker records already provide a certain abstraction and the Kieker-specific Model Manager allows to connect to arbitrary monitoring logs (by design).

We integrated the Java version of Esper [Esper Team and EsperTech, Inc., 2014], which—according to Gualtieri et al. [2009]—is the leading open source CEP engine. On framework startup, an Esper instance is created by the Adaptation Controller. Esper allows to use either an internal or external time source, which is employed to switch between SLAstic's online and offline operation mode.

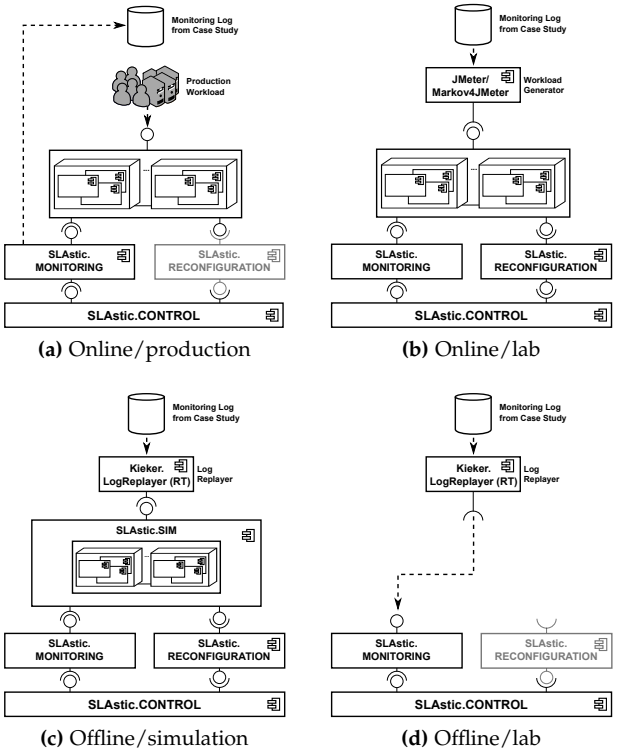


Figure 8.6. Framework uses and integrations for online and offline analyses

8.6.2 Framework Deployments

Figure 8.6 depicts the four typical framework uses and integrations for online and offline analyses.

1. *Online/production (Figure 8.6a).* The SLAstic framework is connected to a production system that is exposed to production workloads. Two modes can be distinguished in this setting. First, SLAstic.Control is used to perform online capacity management. Hence, the framework is used in its intended form. Second, SLAstic.Monitoring logs the observed moni-

toring data into a Kieker Monitoring Log for offline analysis. Particularly, SLAstic.Reconfiguration is not used, e. g., because the system does not provide adaptation capabilities.

2. *Online/lab (Figure 8.6b).* The SLAstic framework is connected to a system in a controlled lab experiment. The system is exposed to synthetic workload derived from logged production workload (previous setting, cf. [Schulz et al., 2014]). A load driver like JMeter/Markov4JMeter [van Hoorn et al., 2008] is used to generate probabilistic and intensity-varying workload. The SLAstic framework is employed for online capacity management.

3. *Offline/simulation (Figure 8.6c).* The SLAstic framework is connected to the SLAstic.SIM simulator for runtime reconfigurable component-based software systems that is triggered by workload events derived from logged production workloads. This setting is detailed in Section 11.3 as part of the description of SLAstic.SIM.

4. *Offline/lab (Figure 8.6d).* The SLAstic framework is connected to the Kieker log replayer that replays Monitoring Records from a Monitoring Log, e. g., collected from a production system. As no adaptable system, neither real nor simulated, is involved, SLAstic.Reconfiguration does not communicate with effector APIs.

8.6.3 Configuration and Startup

The SLAstic framework currently includes a Java API and command-line interface to instantiate and execute a framework in the deployments mentioned in Section 8.6.2. In each case, the class `FrameworkInstance` is instantiated, passing the file system location of the configuration file (Java properties file format). The configuration file specifies, for instance, the concrete types of the framework components to instantiate by providing the fully-qualified class name (cf. Table 8.1) along with component-specific configuration parameters. When calling the framework instance's `run` method, the framework components are instantiated, initialized (properties, component context, etc.), wired, and their `execute` method is called. If desired, a SLAstic.SIM instance is started (cf. Figure 8.6c).

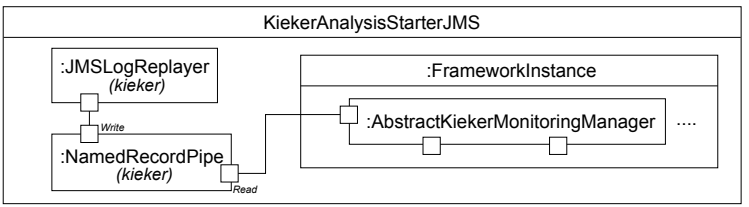


Figure 8.7. Components for online analysis via JMS (cf. Figures 8.6a and 8.6b)

8.6.4 Kieker-based SLAstic Configurations

As previously mentioned, the SLAstic framework builds on and is tightly integrated with the Kieker framework. Particularly, for the SLAstic.Monitoring component, the SLAstic framework already includes concrete implementations that connect to a *Kieker.Monitoring* instance via a JMS or file system *Monitoring Log or Stream* and process incoming *Monitoring Records*. This enables the support for the four online and offline deployment modes described in Section 8.6.2.

In order to invoke a SLAstic framework instance together with a connection to a Kieker *Monitoring Log or Stream*, respective startup tools exist. Figure 8.7 depicts the architecture, exemplified for the connection to a Kieker JMS *Monitoring Stream*. The core components are a Kieker replayer component, which connects to the configured *Monitoring Log or Stream*, and the embedded SLAstic framework instance. In the SLAstic framework instance, this setup requires the use of a Kieker-specific Monitoring Manager component (to extend `AbstractKiekerMonitoringManager`). The abstract Kieker-specific implementation for a Monitoring Manager creates a *Kieker.Analysis* instance, including a *Monitoring Reader* that connects to the JVM-internal Kieker named record pipe also connected to the *Monitoring Writer* as part of the replayer. The *Kieker.Analysis* instance executes asynchronously. Inside the Monitoring Manager, a Kieker pipes-and-filters configuration is created that connects the *Monitoring Reader* with subsequent filters extracting the current time from *Monitoring Records* (Kieker plugin `CurrentTimeEventGenerationFilter`) and passing timer events to the Adaptation Controller's Timer—but only in case the framework is in an offline/replay mode (Figure 8.6d). Concrete Monitoring Manager classes ex-

tending `AbstractKiekerMonitoringManager` refine the prepared pipes-and-filters configuration by additional filters, which perform additional processing of records, e. g., translating the implementation-level monitoring records to architectural monitoring events (Chapter 9). Another form of refinement of the analysis configuration by implementing classes is that the Kieker Monitoring Manager stores incoming *Monitoring Records* as a Kieker file system *Monitoring Log* in the component context for later analysis (Figures 8.6a and 8.6d). For a file system *Monitoring Log*, the respective file system replayer, including a real-time mode, is used. Startup tools for additional types of *Monitoring Logs/Streams* can be implemented and used accordingly.

8.6.5 Concrete Framework Components

During the course of the thesis, concrete framework components, implementing the abstract framework classes in Table 8.1, have been developed. The remainder of this section lists selected implementations, including those for concepts described in later parts of this thesis. These components are part of the publicly available framework implementation.

The Kieker-specific Monitoring Manager has been detailed in Section 8.6.4 already. Accordingly, a Monitoring Manager to connect to SLAstic.SIM (cf. Figure 8.6c and Section 11.3.2) has been implemented in order to receive *Monitoring Records* from the simulator. Model Managers have been developed for the SLAstic meta-model and for the PCM meta-models. Monitoring Manager and Model Updater components implement the *Monitoring Record* transformation and model extraction described in Sections 9.2 and 9.3. A Performance Evaluator component computes and logs time series of selected performance measures, e. g., response times, invocation counts, and resource utilization. A Performance Evaluator that checks SLAs has been developed by Stöver [2009] as part of her thesis. Time- and rule-based Adaptation Planners have been developed for the simulation and lab experiments described in Chapters 13 and 14. Reconfiguration Managers exist to connect the SLAstic framework to the SLAstic.SIM simulator (Section 11.3), to Eucalyptus and AWS infrastructures (Section 13.3), and to reconfigure systems based on HTTP requests [Stöver, 2009].

Chapter 9

Model-Driven Online Capacity Management

This chapter describes model-driven techniques that have been developed as part of the thesis to increase the automation of the SLAstic approach, building on the architectural modeling as well as the Kieker and SLAstic frameworks, described in Chapters 6 to 8. The developed techniques are grouped into the three different parts, namely model-driven instrumentation, transformation of monitoring events, and model extraction via dynamic analysis—summarized below. The three techniques are detailed in the remainder of this chapter in Sections 9.1 to 9.3. Each section includes a description how the respective technique is integrated into the SLAstic framework, including some details about the implementation.

- *Model-Driven Instrumentation.* For building *new* software systems to be equipped with SLAstic's online capacity management support, the recommended choice is to employ a model-driven software engineering (MDSE) approach as introduced in Chapter 2—e. g., according to Stahl and Völter [2006]. The use of MDSE forces architectural models to constitute a representation of the implemented system. Along with implementation artifacts being generated from models expressed using one or more DSLs, system instrumentation can be generated as well. Therefore, we developed an approach for model-driven generation of Kieker-based instrumentation from SLAstic models.

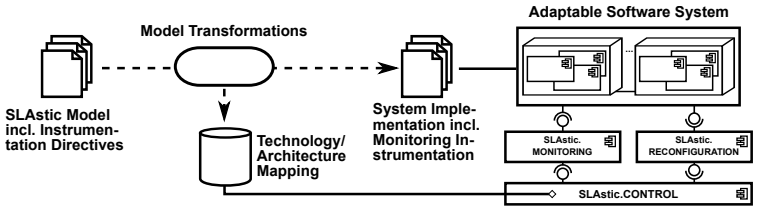
- *Transformation of Monitoring Events.* A core characteristic of the SLAstic framework is the separation between implementation- and technology-aspects of the controlled software system as well as the architectural view on this system employed by the Adaptation Controller to detect,

plan, and execute adaptations (cf. Figure 8.1a). Therefore, we developed a transformation from implementation-level monitoring events to architectural monitoring events.

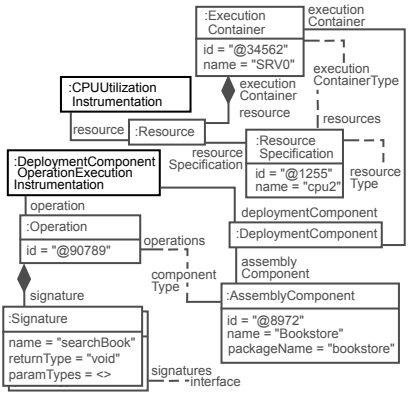
- *Model Extraction via Dynamic Analysis.* SLAstic may also be used with *existing* systems which haven't been built based on MDE. In this case, the software system needs to be instrumented manually in order to provide measurements about its runtime behavior during its execution. Therefore, we developed an approach for reconstructing SLAstic system and usage models (see Chapter 6) from application-level and system-level monitoring data. In a subsequent step, these models can be refined and augmented, e. g., by service level objectives and reconfiguration capabilities.

9.1 Model-Driven Instrumentation

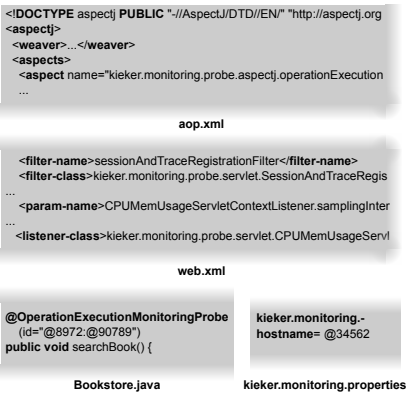
Figure 9.1a depicts the model-driven instrumentation approach. In the ideal case, the system implementation is being developed based on a MDSE-based development process, e. g., as proposed by Stahl and Völter [2006] (cf. Chapter 2). This involves the automatic generation of implementation artifacts, deployment descriptors, etc. based on M2M and M2T transformations, as well as manual implementations by developers. The model-driven instrumentation is integrated into these transformations. As part of the transformation, monitoring instrumentation and configuration is generated based on the instrumentation annotations in the SLAstic model, which have been described in Section 6.5. During the transformation, architectural entities are mapped to implementation-level entities, e. g., component types to Java packages or classes. These mappings are stored in the technology/architecture mapping, which—as described in Section 8.2—is essentially a bijective function that maps architectural identifiers to implementation-level identifiers and vice versa. Note that the examples in this section focus on instrumentation for measuring resource utilization and information about operation executions because these provide measures of interest for this thesis. However, the presented approach is not limited to these types of instrumentation but can be extended accordingly. Also, we focus on Kieker-based instrumentation, which is not a conceptual constraints but provides a defined target of generation.



(a) Overview of model-driven instrumentation approach



(b) SLAstic model (excerpt) including two monitoring directives



(c) Example implementation artifacts including generated Kieker monitoring instrumentation and configuration (excerpts)

Figure 9.1. Overview and examples of model-driven instrumentation approach

Figures 9.1b and 9.1c depict an excerpt of a SLAstic model for the running example including architecture-level instrumentation directives, as well as example excerpts from generated technology-specific monitoring instrumentations and configurations for the Kieker framework respectively.

- The two instrumentation directives included in Figure 9.1b, which were introduced in Section 6.5, indicate that monitoring probes are to be added to the software system, which measure the utilization of the referenced resource (CPU in this case) as well as information about executions of the referenced `Operation` of the referenced `DeploymentComponent`.

Table 9.1. Kieker record types and corresponding SLAstic meta-classes

Kieker record type	SLAstic event type
OperationExecutionRecord	*ConnectorOperationExecution*, *DeploymentComponentOperationExecution*
ResourceUtilizationRecord	*ResourceUtilization*
CPUUtilizationRecord	*CPUUtilization*
MemSwapUsageRecord	*MemSwapUsage*

- Figure 9.1c sketches possible Kieker-specific monitoring instrumentation and configuration generated from the architecture-level instrumentation directives. For this example, we assume that instrumentations for operation executions are added based on Java annotations processed by AspectJ [Kiczales et al., 2001] (Section 4.2.2). This instrumentation mechanism is included in the Kieker implementation (see [Kieker Project, 2014a]). In this case, the AspectJ configuration file (*aop.xml*) needs to be generated, which includes the list of aspects to use. A Java annotation is added to each method to be monitored (example in *Bookstore.java*). In the example, the annotation includes implementation-level identifiers to be included in the *Monitoring Records* produced by this *Monitoring Probe*. The Kieker configuration file (*kieker.monitoring.properties*) includes the identifier of the host name. The CPU utilization is monitored by a Kieker *Monitoring Probe* integrated based on Java Servlet technology (Section 3.3.1, registered in the application's *web.xml*). Note that a complementary Servlet-based *Monitoring Probe* supports the monitoring of operation executions (*web.xml*).

Note that in this case, we assume a 1:1 mapping function between architectural identifiers and implementation-level identifiers included in Figures 9.1b and 9.1c.

9.2 Transformation of Monitoring Events

As described in Section 8.3, the Monitoring Manager transforms implementation-level monitoring events to architectural monitoring events. This section details this transformation. Implementation-level monitoring events are instances of Kieker monitoring records (Figure 7.2). Architectural

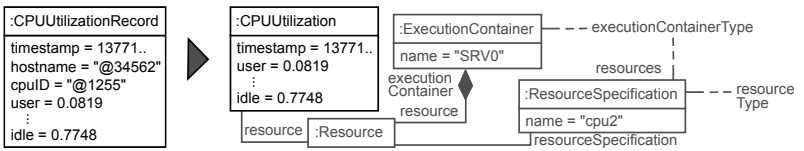


Figure 9.2. Transformation result for *CPUUtilizationRecord*. Existing entities are displayed in gray color.

monitoring events are instances of the SLAstic monitoring events, which are part of the SLAstic meta-model (Section 6.3.1). Table 9.1 lists the Kieker monitoring records and SLAstic event types that are considered in this section, as these are relevant for this thesis. The meta-classes are included in Figures 6.8 and 7.2. In case additional monitoring events are added, respective transformations need to be developed. Note that for the transformations described in this section, we assume that all model entities that are looked up already exist. However, on-demand creation of entities is supported by including the transformation for model extraction, which will be described in Section 9.3.

Sections 9.2.1 and 9.2.2 detail the transformation of monitoring events for resource usage (CPU, generic resources, and memory) and operation executions. Java-based implementations of the transformation are included in the SLAstic framework implementation described in Section 8.6 as part of the Monitoring Manager (SLAstic.Monitoring).

9.2.1 Resource Usage

Transformations of resource usage monitoring events currently exist for utilization of CPU and generic resources, as well as for memory usage. These transformations are detailed in the remainder of this section.

CPU Utilization

This transformation takes a Kieker *CPUUtilizationRecord* as input and creates a SLAstic *CPUUtilization* event as output. An example is depicted in Figure 9.2. The easy part of the transformation is the assignment of the properties with plain type, i. e., the timestamp and the utilization (*combined*,

idle, *irq*, *nice*, *system*, *user*, *wait*) values by the corresponding values of the *CPUUtilizationRecord*. The *Resource* to be referenced by the *CPUUtilization* event is determined in a two-step procedure. First, the *ExecutionContainer* is looked up using the Model Manager: *a*) looking up the corresponding architectural identifier for the host name's implementation-level identifier given by the *CPUUtilizationRecord's hostname* property, and *b*) looking up the *ExecutionContainer* using the architectural identifier. In a second step, the *Resource* is retrieved from the Model Manager in a similar procedure: *a*) looking up the architecture-level identifier for the resource identifier given by the *CPUUtilizationRecord's cpuID* property, and *b*) looking up the *Resource* with this architectural identifier to be contained in the *ExecutionContainer*. Note that in this case, the *Resource* has the resource type *CPUType*.

Generic Resource Utilization

This transformation takes a Kieker *ResourceUtilizationRecord* as input and creates a SLAstic *ResourceUtilization* event as output. The procedure of this transformation equals the afore-described transformation. First, the timestamp and utilization values are assigned. In the second step, the matching *Resource* is retrieved from the Model Manager, as described for the CPU utilization event transformation. Note that in this case, the implementation-level resource identifier is contained in the *ResourceUtilizationRecord's* property *resourceName* and that the retrieved is of resource type *GenericResourceType*.

Memory Usage

This transformation takes a Kieker *MemSwapUsageRecord* as input and creates a SLAstic *MemSwapUsage* event as output. Again, this transformation is a two-step procedure: *a*) assigning the values of plain type (*memUsedBytes* and *swapUsedBytes*) and *b*) retrieving and assigning the appropriate *Resource*. Note that the *MemSwapUsageRecord's* properties about the total amount of memory and swap (*memTotal* and *swapTotal*) are not considered. The retrieved *Resource* has the type *MemSwapType* and references a *MemSwapResourceSpecification*.

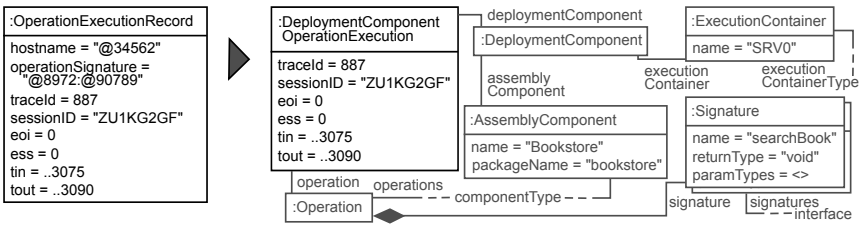


Figure 9.3. Transformation result for *OperationExecutionRecord*. Existing entities are displayed in gray color.

9.2.2 Operation Executions

This transformation takes a Kieker *OperationExecutionRecord* as input and creates a SLAstic *OperationExecution* event as output, which is either of concrete type *DeploymentComponentOperationExecution* or *ConnectorOperationExecution*. Figure 9.3 depicts an example. According to the transformations described in Section 9.2.1, this transformation can be separated into two parts: *a*) assignment of plain values and *b*) lookup and assignment of references to entities in the SLAstic runtime model. The simple part—assignment of values—concerns the properties *traceId*, *sessionId*, *eoi*, *ess*, *tin*, and *tout*, which are present in both objects. Additionally, a *DeploymentComponentOperationExecution* includes references to a *DeploymentComponent* and an *Operation*, which need to be looked up employing the Model Manager. The basis for looking up the *DeploymentComponent* are the implementation-level identifiers contained in the *OperationExecutionRecord's* properties *hostname* and *operationSignature*. Using the host name identifier, the corresponding *ExecutionContainer* is looked up as described for the transformation of CPU utilization records. As exemplified in Figure 9.1, the property *operationSignature* is a concatenation of two identifiers—(implementation-level) component and operation. The implementation-level component identifier is used to lookup the corresponding *AssemblyComponent*, after having translated the implementation-level identifier to the architecture-level identifier. The looked up *ExecutionContainer* and *AssemblyComponent* are now used to retrieve the *DeploymentComponent* to be assigned to the resulting *DeploymentComponentOperationExecution*. The *Operation* is looked up from the Model Manager using the *ComponentType* and the (translated) operation identifier.

9.3 Model Extraction via Dynamic Analysis

In order to support scenarios in which SLAstic models for the controlled software system are not available and in which no model-driven instrumentation according to Section 9.1 is employed, this section details the automatic extraction approach based on dynamic analysis. The extraction concerns the system structure and its usage, represented by the respective parts of the SLAstic meta-model covered in Sections 6.2 and 6.3. The extraction process is integrated in the following three phases: *a*) transformation of monitoring events including the on-demand creation and updating of architectural entities, such as types, (assembly and deployment) components, and execution containers; *b*) trace reconstruction based on operation execution events; *c*) processing of the reconstructed traces for updates of the usage model (e. g., call relationships and frequencies), assembly model (e. g., assembly connectors), as well as the execution environment (e. g., network links). The first phase is integrated into the transformation approach for monitoring events described in Section 9.2. In this case, the mapping function including the relation of implementation-level entities to architectural entities is created and updated on-the-fly. Note that the usage model extraction is also relevant for scenarios in which SLAstic models are available and model-driven instrumentation is used, because workloads are typically varying over time (Section 4.4).

Section 9.3.1 describes the extraction of control flow traces and the usage model. Section 9.3.2 details the on-demand creation and refinement of architectural entities. The type and operation signature abstraction is detailed in Section 9.3.3. A Java-based implementation of the described extraction is included in the SLAstic framework implementation described in Section 8.6 as part of the Monitoring Manager (SLAstic.Monitoring) and the Model Updater (SLAstic.Control).

9.3.1 Trace and Usage Model Extraction

This section describes how traces are reconstructed from the stream of SLAstic `OperationExecution` events and how these traces are processed to update the SLAstic usage model. For details on the SLAstic meta-model parts for traces and the usage model, please refer to Section 6.3. Note

```
select a, b from pattern
[
  every-distinct(a.traceId, TIMEOUT+1 sec)
  a=OperationExecution
  ->
  ((b=OperationExecution(traceId=a.traceId)
    OR
    time:interval(TIMEOUT sec))
   until (timer:interval(TIMEOUT+1 sec) OR TerminationRecord)
  )
]
```

Figure 9.4. EPL statement to collect operation executions of a trace. The statement is parameterized by a `TIMEOUT` variable.

that for this step it is assumed that SLAstic *`OperationExecution`* events are available, e. g., by the transformation described in Section 9.2.2 that may be extended by the on-demand creation that will be described in Section 9.3.2.

Trace Reconstruction

Similar to the trace reconstruction approach implemented in the Kieker framework (Section 7.2), this reconstruction of traces is based on collecting operation executions with an equal trace identifier and reconstructing the calling relationships based on the control flow information contained in the monitoring events (*`eoi`* and *`ess`* properties).

First, the SLAstic's CEP Engine is used to group operation executions by their trace identifiers. Operation executions with an equal trace identifier are assumed to belong to the same trace. Figure 9.4 shows the pattern-based EPL query that collects groups of executions with the same trace identifier (*`traceId`* property) from the incoming stream of *`OperationExecution`* events. The CEP Engine instantiates a new pattern when it observes a trace identifier that it has not observed before within the given period of time (`TIMEOUT` + 1 seconds). The *`OperationExecution`* triggering the instantiation of this pattern is the first element in the result set for this query—being

an array of *OperationExecution* objects with one or more elements. Now the CEP Engine awaits a sequence of additional *OperationExecutions* with this trace identifier. The query terminates and returns its result set after no additional matching event has been observed for a given period of time (based on the TIMEOUT variable) or the SLAstic framework terminates (*TerminationRecord* event). For a more detailed description on pattern-based queries in EPL, please refer to the EPL documentation [Esper Team and EsperTech, Inc., 2014].

In a second step, the set of *OperationExecutions* is further processed in a trace reconstruction step, resulting in a *Trace* object (Figure 6.9). The *OperationExecutions* are sorted by *eoi* and transformed into a *MessageTrace* (Figure 6.9) by emulating a stack machine using the *OperationExecution's* *eoi* and *ess* values. In case the set of *OperationExecutions* is incomplete, the trace reconstruction fails and the resulting *Trace* object is of concrete type *InvalidTrace*. In the case of success, the concrete type is *MessageTrace*, which is a *ValidTrace*.

Trace Processing for Usage Model Updates

Further processing of traces is performed only on *MessageTrace* objects, as instances of a *ValidTrace*; i. e., *InvalidTrace* objects are not considered. A *MessageTrace* is processed by iterating over the sequence of contained *Messages*. The remainder of this section describes the processing algorithm.

For each *Message* m, the algorithm first determines whether it is a *SynchronousCallMessage* or a *SynchronousReplyMessage*.

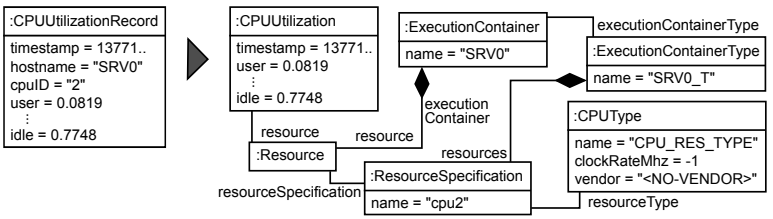
- In case m is a *SynchronousCallMessage*, it increments the call frequency of the called *Operation* in the usage model (*OperationCallFrequency*) by 1.
- In case m is a *SynchronousReplyMessage*, it first determines the provided *Interface* of the returning *OperationExecution* from the Model Manager based on the *ComponentType* and *Signature* associated with the executed *Operation*. If m is a reply message to a system entry call (i. e., not originating from a system-internal call), the *SystemProvidedInterfaceDelegationConnector* involved in this execution is looked up (based on the providing *AssemblyComponent* and the *Signature*) and the calling frequency of this *Signature* is incremented by 1 for this connector. Otherwise, i. e., if m is associated with a system-internal call, the algorithm performs two steps.

First, according to the system entry call, the `AssemblyComponentConnector` is looked up (based on the requiring and providing `AssemblyComponents` and the `Signature`) and the calling frequency of the `Signature` is incremented by 1 for this connector. Second, the count of how many times the calling `Operation` (associated with the receiver of m) calls the `Signature` of the currently involved required `Interface` is updated. Note that this information is further updated while processing this trace and propagated to the Model Manager after the entire trace is processed.

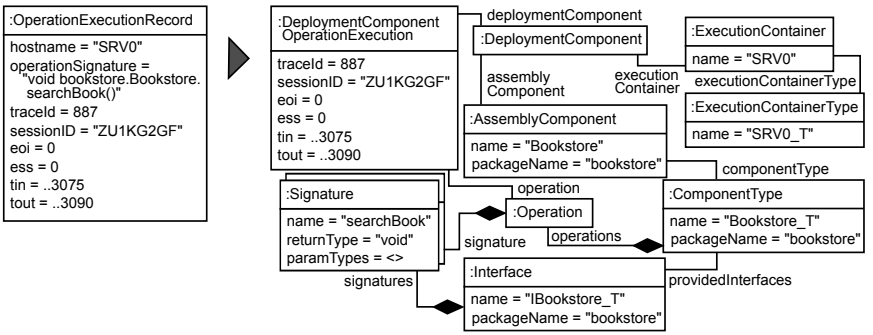
After each `Message` contained in the `Trace` has been processed, the frequency distribution of calling relationships for calls from each operation to signatures of required interfaces is propagated to the Model Manager for being updated in the usage model. Remember that the invocation count to each called interface/signature combination from each executed operation was collected and updated while processing the trace.

9.3.2 On-Demand Creation and Refinement of Architectural Entities

This section details the on-demand creation and refinement of architectural entities. The use case is that the query for a mapping of an architectural entity to an implementation-level entity results in an empty result. In this case, missing entities are created on-the-fly and the mapping is created (with a newly perceived architectural identifier) so that future lookups lead to a valid result (previously created entity). This section focuses on the on-demand creation capabilities for the afore-mentioned transformation from Kieker monitoring records to SLAstic monitoring events and the usage model extraction. The set of discovery functions may need to be extended for additional transformations and extractions. Note that this section covers only the SLAstic system model, as the update of the usage model has already been covered in Section 9.3.1. The description is structured into five parts based on the covered entity types, namely *a*) execution container and execution container types, *b*) resources, *c*) components, component types, and interfaces, *d*) operations, as well as *e*) assembly connectors, required interfaces, and network links.



(a) Transformation of *CPUUtilizationRecord*



(b) Transformation of *OperationExecutionRecord*

Figure 9.5. Transformation results for *CPUUtilizationRecord* (a) and *OperationExecutionRecord* (b) including the on-demand creation of architectural entities

▻ *Execution Container and Execution Container Type*

The on-demand creation operation for an *ExecutionContainer* expects a name for the entity to be created. This is typically the implementation-level host name contained in the Kieker record (*hostname* property). Included in the on-demand creation is the creation of an *ExecutionContainerType*. The name used for the *ExecutionContainerType* is the *ExecutionContainer* name extended by a type suffix ("_T"). Both the newly created *ExecutionContainer* and the corresponding *ExecutionContainerType* contain no specifications of resources. The new *ExecutionContainer* is marked allocated.

As an example, Figures 9.5a and 9.5b include the on-demand creation of an *ExecutionContainer* and a corresponding *ExecutionContainerType* for the host name "SRV0" contained in the transformed Kieker records.

▷ *Resources*

The on-demand creation operation for a *Resource* expects a *Resource* name and *Resource* type identifier, as well as the *ExecutionContainer* to contain the *Resource*. The operation first determines whether a *ResourceSpecification* with this name already exists in the corresponding *ExecutionContainerType*. If this is not the case, a *ResourceSpecification* with the *Resource* name is created for the *ExecutionContainerType*. In order to avoid name clashes among the different types of resources, a type-specific prefix is added to the *Resource* name, e. g., "cpu". Along with the *ResourceSpecification*, an appropriate *ResourceType* is created, e. g., *CPUType* for CPU resources. Default values are assigned to the *ResourceType* in case no detailed information is available, e. g., the clock rate of a CPU. Note that for a memory resource, a *MemSwapResourceSpecification* is created; the information about memory and swap size are included in a Kieker *MemSwapUsageRecord*.

As an example, Figure 9.5a includes the on-demand creation of a *Resource*, a *ResourceSpecification*, and a *CPUType* for the implementation-level CPU resource on the *ExecutionContainer* "SRV0".

▷ *Components, Component Types, and Interfaces*

The on-demand creation of an *AssemblyComponent* expects a name for the entity to be created. The basis for this name is, for instance, the implementation-level class name contained as part of the *operationSignature* property in Kieker's *OperationExecutionRecords*, which is further processed by the procedure described in Section 9.3.3. Included in the creation of an *AssemblyComponent* is the creation of a *ComponentType* and an *Interface*. As for the on-demand creation of an *ExecutionContainer* and a corresponding *ExecutionContainerType*, the name of the *ComponentType* is the *AssemblyComponent* name extended by a type suffix ("_T"). The *Interface* name includes an additional prefix "I". The new *Interface*, which contains no *Signature* so far, is added to the *ComponentType's* list of provided *Interfaces*. The on-demand creation of a *DeploymentComponent* follows the afore-described on-demand creation of an *ExecutionContainer* and an *AssemblyComponent*—along with their associated entities.

As an example, Figure 9.5b includes the on-demand creation of a *DeploymentComponent*, an *AssemblyComponent*, a *ComponentType*, and an *Interface*.

▷ *Operations*

The on-demand creation of an *Operation* expects a *ComponentType* and an operation signature. The operation signature particularly includes an operation name, parameter types, and a return type. The basis for these values is, for instance, the implementation-level signature contained as part of the *operationSignature* property in Kieker's *OperationExecutionRecords*, which is further processed by the procedure described in Section 9.3.3. A *Signature* with the respective values is created and assigned to the new *Operation*. The *Operation* is added to the *ComponentType's* list of operations. Moreover, another equal *Signature* is created and added to the *ComponentType's* default *Interface*.

As an example, Figure 9.5b includes the on-demand creation of an *Operation* and two (equal) *Signatures*.

▷ *Assembly Connectors, Required Interfaces, and Network Links*

The on-demand creation of *AssemblyConnector*s is part of processing a *SynchronousReplyMessage* of a *Trace*. In case, the message refers to a reply for an entry call, this may involve the creation of a *SystemProvidedInterfaceDelegationConnectors*; otherwise, i. e., no entry level call, an *AssemblyComponentConnectors* may need to be created. A *SystemProvidedInterfaceDelegationConnector* is created based on the *Interface* and the *AssemblyComponent* (the delegation receiver) involved in the reply. Similarly, an *AssemblyComponentConnector* is created based on the *Interface* as well as the calling and receiving *AssemblyContexts* involved in the reply message. For the requiring *AssemblyContext*, this can involve the addition of the *Interface* to the list of required *Interfaces*. A *NetworkLink* between the *ExecutionContainers* involved in the reply message is created, in case it does not exist.

9.3.3 Type and Operation Signature Name Abstraction

During on-demand creation of architectural entities, these entities are assigned names that are based on the implementation-level names found in

the Kieker monitoring records. For example, an *OperationExecutionRecord* has the property *operationSignature* that comprises the fully-qualified implementation type name, as well as the implementation-level operation name, parameter and return types, modifiers, etc. (see Section 7.2). Intuitively, one could map this implementation-level type and signature to architectural entity names as part of the on-demand creation (Section 9.3.2). However, typically implementation-level names do not have a 1:1 relationship to architectural names. Instead, an implementation-level package may correspond to an architectural component. For this reason, this section describes our approach for type and operation signature name abstraction used during on-demand creation of architectural entities. Note that this approach is only needed for on-demand creation of entities and not if a model-driven instrumentation approach, as described in Section 9.1, is employed. In that case, architectural entities are determined based on the architecture/technology mapping maintained by the Model Manager. In this section, we will present four different abstraction modes. Additional modes can be added easily.

Name Abstraction Based on Package Hierarchy

The basic idea is that a 3-tuple `(package, type, operation)` of fully-qualified implementation-level type name (split into `package` and `type`) and operation signature (`operation`) is transformed into a 3-tuple `(package', type', operation')`, which includes the abstracted architecture-level names. The approach is based on a name hierarchy's depth d. For a fully-qualified name, the parameter d specifies at what level in the name hierarchy the package name ends and the type name starts. For example, consider the implementation-level tuple `("a.b.c", "D", "void op()")`. On implementation-level, the fully-qualified name has a depth $d = 4$ because the type name is the fourth element in the hierarchy. Assuming an abstraction mode that maps elements at level 3 to architectural entities, the resulting 3-tuple would be `("a.b", "C", "void d_op()")`. In this case, the implementation-level class name `"D"` becomes prefix of the operation signature.

Table 9.2. Example results of type and operation signature name abstraction

Input	d	Output		
		Mode 1 *(Class strict)*	***Mode 2*** *(Package strict)*	***Mode 3*** *(Single component)*
A.op(...)	1	A.op(...)	N/A	@.a_op(...)
a.B.op(...)	2	a.B.op(...)	A.b_op(...)	@.a_b_op()
a.b.C.op(...)	3	a.b.C.op(...)	a.B.c_op(...)	@.a_b_c_op(...)
a.b.c.D.op(...)	4	a.b.c.D.op(...)	a.b.C.d_op(...)	@.a_b_c_d_op(...)
		Mode 4 (l=1) *(Package level)*	***Mode 4 (l=2)*** *(Package level)*	***Mode 4 (l=3)*** *(Package level)*
A.op(...)	1	A.op(...)	@.a_op(...)	@.a_op(...)
a.B.op(...)	2	A.b_op(...)	a.B.op(...)	@.a_b_op(...)
a.b.C.op(...)	3	A.b_c_op(...)	a.B.c_op(...)	a.b.C.op(...)
a.b.c.D.op(...)	4	A.b_c_d_op(...)	a.B.c_d_op(...)	a.b.C.d_op(...)

Abstraction Modes

Table 9.2 includes examples for the four different abstraction modes that will be described below. Operation modifiers, return types, and parameter types are omitted from these and the following examples for the sake of comprehensibility. The 3-tuples are given in the typical serialized form for operation signatures (see also Section 7.2) with the three tuple elements being separated by a dot ("."). The parameter type list is depicted by `"(...)"`. The four abstraction modes work as follows:

1. *Mode 1: Class strict.*

   This mode implements the naïve approach of mapping implementation-level package, type, and signature names directly to the respective names on the architectural level. For example, the input 3-tuple `("a.b.c", "ClassD", "op(...)")` results in the equal output 3-tuple `("a.b.c", "ClassD", "op(...)")`. See Table 9.2 for additional examples.

2. *Mode 2: Package strict.*

   This mode assumes that the last element of an implementation-level package name maps to the architectural type. For example, the implementation-level 3-tuple `("a.b", "ClassC", "op(...)")` results in the 3-

tuple `("a", "B", "classC_op(...)")`; the 3-tuple `("a.b.c.d", "ClassE", "op(...)")` results in `("a.b.c", "D", "classE_op(...)")`. Note that the capitalization of names is modified according to common naming conventions. Name elements that become part of the signature name are separated by `"_"` (dots being replaced by a single `"_"` character). See Table 9.2 for additional examples.

3. *Mode 3: Single component.*

   This mode assumes that all implementation-level types correspond to a single architectural type. For example, the 3-tuple `("a.b.c", "ClassD", op(...)")` results in the 3-tuple `("", "@", "a_b_c_theClass_op(...)")`, with `"@"` denoting a fix type name. See Table 9.2 for additional examples.

4. *Mode 4: Package level.*

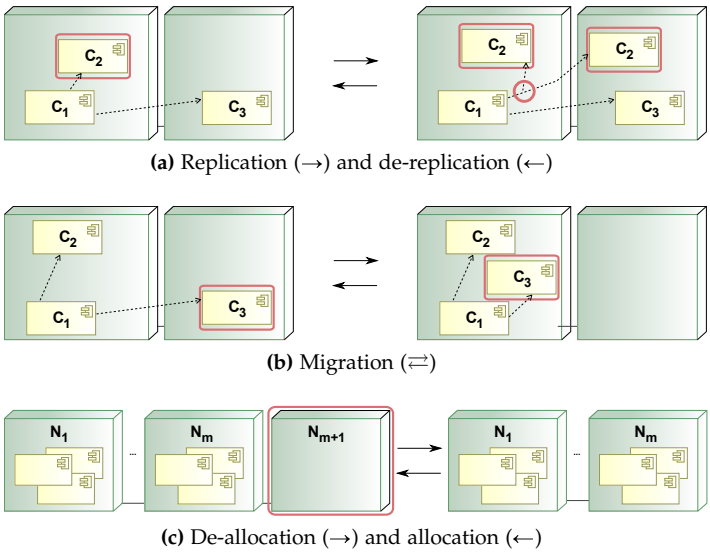
   In this mode, which is parameterized by a parameter $l \in \mathbb{N}^+$, the name element at level $l \leqslant d$ becomes the type. For example, the 3-tuple `("a.b.c", "ClassD", "op(...)")` ($d = 4$) results in the output 3-tuple `("a.b", "C", "classD_op(...)")` for $l = 3$. For $l \geqslant d + 1$, the result equals the result from mode 3 (single component). See Table 9.2 for additional examples.

Chapter 10

Runtime Reconfiguration for Controlling Capacity

In the SLAstic approach, architectural runtime reconfiguration, as introduced in Section 3.4.2, is the means to apply architecture-based online adaptation to the controlled system. The SLAstic meta-model includes concepts to express reconfiguration-related properties of the system, including reconfiguration operations and plans, as well as reconfiguration capabilities and properties (Section 6.4). The SLAstic framework (Chapter 8) includes components to plan and execute adaptations on an architectural level in form of reconfiguration plans, interacting with respective technology-specific effectors provided by the controlled software system. In principle, the SLAstic allows to integrate arbitrary runtime reconfiguration operations that are based on architectural entities from the SLAstic meta-model, particularly w.r.t. the system structure (Section 6.2). This chapter focuses on the five runtime reconfiguration operations that serve as a means for online capacity management in this thesis. The description aims to serve as a blueprint on how to integrate additional operations.

This chapter is structured as follows. Section 10.1 provides a first overview of the operations. Section 10.2 details their integration into the SLAstic framework, including extensions to the meta-model, the Model Manager, and the Reconfiguration Manager. Note that technology-specific implementations of these operations will be provided and used in Section 11.3.3 as well as Chapters 13 and 14. This chapter contains contents from our previous publications [van Hoorn et al., 2009a,b; von Massow et al., 2011; Huber et al., 2014].



(a) Replication (→) and de-replication (←)

(b) Migration (⇄)

(c) De-allocation (→) and allocation (←)

Figure 10.1. SLAstic reconfiguration operations [von Massow et al., 2011]

10.1 Overview of Operations

The scope of this thesis is architecture-based online capacity management. Based on architectural knowledge about the overall system and its usage, the architectural configuration is changed at runtime in order to provide adequate capacity. As introduced in Section 3.4.2, architecture-based online adaptation employs architectural runtime reconfiguration as a means to carry out change to the controlled software system. In this thesis, we are particularly interested in those changes to a software system that have an impact to its measure of adequate capacity. Based on Menascé and Almeida's definition of adequate capacity (Definition 4.2 on page 43), capacity concerns two main system properties, namely SLAs and costs. For this thesis, we decided to focus on a set of architectural reconfiguration operations having an impact on a system's capacity.

The considered runtime reconfiguration operations are illustrated in Figure 10.1. They comprise three application-level operations—software com-

ponent replication, de-replication, and migration (Figures 10.1a and 10.1b)—and two system-level operations—execution container allocation and de-allocation (Figure 10.1c). We will refer to these five operations as the *SLAstic runtime reconfiguration operations* in this thesis. A UML-like graphical concrete syntax is used in Figure 10.1 to represent SLAstic model instances—in this case showing parts of the component assembly model, the component deployment model, as well the execution environment model. For example, the right part of Figure 10.1a includes a SLAstic instance that includes three assembly components, C_1, C_2, and C_3, which are deployed to two execution containers. A single deployment component exists for both C_1 and C_3. For C_2, two deployment components exist—being distributed over the two execution containers. The arrows between the deployment components refer to requires/provides relationships of the assembly components defined in the component assembly model. In this case, C_1 requires the assembly components C_2 and C_3. In this example, calls from C_1 to C_3 are remote calls. Calls from $C1$ to C_2 are distributed between the two deployment components of C_2.

Sections 10.1.1 to 10.1.5 describe the five SLAstic runtime reconfiguration operations, which aim to control a system's properties with respect to capacity. We will informally describe the semantics of these operations and their expected impact on system capacity. The integration of these operations into the framework will be described in Section 10.2. Note that each of the operations requires a technology-specific implementation, which makes use of the actual system's effector APIs via a respective Reconfiguration Manager. However, we provide a high-level description of the expected semantics. The description is based on the assumptions about transactional reconfiguration of CBSSs, as proposed by Matevska [2009].

10.1.1 Software Component Replication

The replication operation creates an additional deployment component for an assembly component on an execution container, which is already deployed to the execution environment model. In the example in Figure 10.1a (from left to right), a second deployment component of C_2 is instantiated. As described above, calls to C_2 are distributed between the two deployment components. The strategy how calls are distributed is implementation-specific, e. g., calls by the same client may always be distributed to the

same deployment component. Note that each execution container must not contain more than one deployment component for the same assembly component.

The goal of applying this reconfiguration operation is to distribute the amount of resources used by C_2 to other—already-allocated—execution containers. Coming back to the example (Figure 10.1a), possible triggers for executing this reconfiguration operation are that the workload demands to C_1 and/or C_2 increase, requiring additional capacity for these services. In this case, capacity on the other execution container, which may not be fully utilized, is available. This may decrease response times for C_1 and C_2, which may be necessary to satisfy SLAs. With respect to costs, applying this reconfiguration operation may increase resource efficiency by utilizing allocated resources more efficiently.

10.1.2 Software Component De-Replication

The de-replication operation is the inverse operation to the afore-described replication. A deployment component is removed from the component deployment model. Note that it is required, that at least another deployment component for the same assembly component exists. In the example in Figure 10.1a (from right to left), deployment component C_2 is removed. Again, the implementation of this operation is technology-specific. However, a common approach, based on the proposed strategy for the replication, is that the deployment component to be removed is blocked in a sense that requests by new clients are not distributed to this deployment component. The deployment component can be safely removed after all requests by clients using that component are served (cf. [Matevska, 2009]).

The goal of applying this reconfiguration is to release capacity used by the component to be de-replicated. Possible triggers in the example could be that the workload demands to C_1 and/or C_3 decrease and that capacity provided by a single deployment component of C_2 is sufficient. Moreover, the application of this operation can be a preparation for applying an execution container de-allocation operation, as detailed below.

10.1.3 Software Component Migration

Conceptually, the migration operation moves a deployment component from one execution container to another. Figure 10.1b includes two applications of the migration operation to C_3, namely moving it from a dedicated execution container to an execution container shared with C_1 and C_2 (from left to right), as well as the reverse direction (from right to left). In certain cases, this operation may be implemented as a combination of the afore-described replication and the de-replication operations. However, on the architectural level, it makes sense to have an explicit migration operation. Note that for stateful components, the application of this operation may involve a migration of state.

The goals for applying this operation for capacity management correspond to the goals of the replication and de-replication operations, e. g., enabling a more efficient use of system resources by using already-allocated resources or freeing resources, which can be released in a subsequent step.

10.1.4 Execution Container Allocation

The allocation operation adds an execution container to the list of allocated execution containers in the execution environment model. This system-level operation consists of the allocation of a (virtual or physical) server node, which may include the installation of middleware services or Application Servers. The example in Figure 10.1c (from right to left) includes the allocation of the execution container N_{m+1}. After applying this operation, N_{m+1} is, for example, available as a target of software component replication or migration operations. Intuitively, the goal of the allocation is the provisioning of additional capacity in terms of computing resources.

10.1.5 Execution Container De-Allocation

The de-allocation operation is the reverse operation of the allocation operation, i. e., it removes an execution container from the list of allocated execution containers in the execution environment model. A precondition for applying this operation is that no deployment component is located on the respective execution container. The example in Figure 10.1c (from left to right) includes the de-allocation of the execution container N_{m+1}.

Operating costs—e. g., caused by power consumption or usage fees in cloud environments—can be saved by de-allocating execution containers.

10.2 Framework Integration

This section describes how the five SLAstic reconfiguration operations are integrated into the framework, namely by extending the SLAstic meta-model (Section 10.2.1), the Model Manager (Section 10.2.2), as well as the Reconfiguration Manager (Section 10.2.3). The Model Manager is responsible for reflecting the changes into the runtime model based on model transformations. The Reconfiguration Manager is responsible for executing the architectural operations employing the technology-specific effector APIs. Note that the description of the framework integration also serves as a blueprint for integrating additional runtime reconfiguration operations into the SLAstic framework.

10.2.1 Meta-Model Extensions

The SLAstic meta-model includes reconfiguration-specific modeling constructs (Section 6.4), namely adaptation operations, adaptation capabilities, and adaptation plans. For each reconfiguration operation, a meta-class has to be created that extends the abstract meta-class `ReconfigurationAction` (Section 6.4.1). Each concrete `ReconfigurationAction` class basically declares the signature of the reconfiguration operation, i. e., including the operation's name (which is the meta-class name) as well as its parameters. We will omit the description of extensions for adaptation capabilities. Extensions w.r.t. the adaptation plan meta-modeling are not required.

For the five SLAstic runtime reconfiguration operations, Figure 10.2 shows the respective meta-classes in S/T/A notation—each action corresponds to an S/T/A `Action`; corresponding `ReconfigurationAction` meta-classes exist.

- The S/T/A action `replicateComponent` (`ReconfigurationAction` meta-class `ComponentReplication`) corresponds to the software component replication operation from Section 10.1.1. It expects an `AssemblyComponent` (`component`) and an `ExecutionContainer` (`to`) as inputs, and returns a `DeploymentComponent`

(*component*). The expected result is that after the successful execution of the operation, the returned model entity refers to an additional deployment component of the input assembly component, which is deployed to the given execution container. Note that as a precondition, the execution container must be allocated and must not contain a deployment component for the given assembly component (cf. Section 10.1.1).

- The S/T/A action *dereplicateComponent* (*ReconfigurationAction* meta-class *ComponentDereplication*) corresponds to the software component de-replication operation from Section 10.1.2. It expects a *DeploymentComponent* (*component*) as input. The expected result is that after the successful execution of the operation, the given deployment component does no longer exist. Note that as a precondition, another deployment component for the corresponding assembly component must exist (cf. Section 10.1.2).
- The S/T/A action *migrateComponent* (*ReconfigurationAction* meta-class *ComponentMigration*) corresponds to the software component migration operation from Section 10.1.3. It expects a *DeploymentComponent* (*component*) and an *ExecutionContainer* (*to*) as inputs, and returns a *DeploymentComponent* (*component*). The expected result is that after the successful execution of the operation, the given deployment component no longer exists and instead a new deployment component for the common assembly component

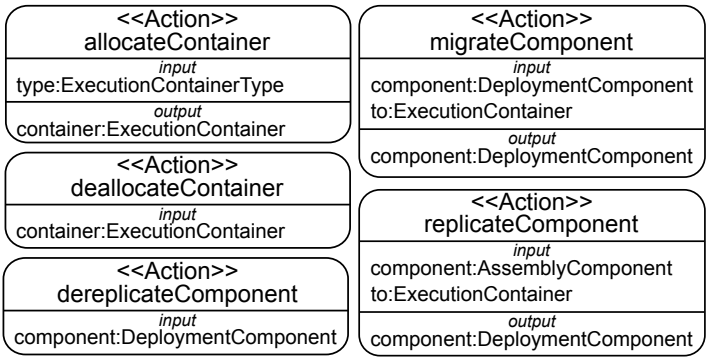


Figure 10.2. S/T/A actions corresponding to the SLAstic runtime adaptation operations, including the list of input and output parameters [Huber et al., 2014]

exists on the given execution container. Note that as a precondition, the execution container must be allocated and must not contain a deployment component for the given assembly component (cf. Section 10.1.3).

- The S/T/A action *allocateContainer* (*ReconfigurationAction* meta-class *ContainerAllocation*) corresponds to the execution container allocation operation from Section 10.1.4. It expects an *ExecutionContainerType* (*type*) as input. The expected result is that after the successful execution of the operation, a new execution container of the given type is available.
- The S/T/A action *deallocateContainer* (*ReconfigurationAction* meta-class *ContainerDellocation*) corresponds to the execution container de-allocation operation from Section 10.1.5. It expects an *ExecutionContainer* (*container*) as input. The expected result is that after the successful execution of the operation, the given execution container is no longer allocated. Note that as a precondition, no deployment component must be located on this execution container (cf. Section 10.1.5).

10.2.2 Model Manager Extensions

As described in Section 8.2.7, one of the Reconfiguration Model Manager's responsibilities is the reflection of executed reconfigurations in the SLAstic model. In order to do this, it must provide respective operations to trigger model transformations. The respective part of the Model Manager includes operations to execute changes to the runtime model. Already during the execution of a reconfiguration plan by the Reconfiguration Manager, the Model Manager updates the runtime SLAstic model using the other submodel managers.

Note that for the five considered SLAstic runtime reconfiguration operations, the type repository model and the component assembly model are read-only, i. e., no changes are executed in these models. Changes are only executed in the execution environment model and the component deployment model.

Table 10.1 lists the Reconfiguration Model Manager's relevant operation signatures. The Reconfiguration Model Manager only executes the operations if the respective preconditions, which were mentioned in Section 10.2.1, are satisfied. The performed model transformations are as follows (imperative description):

Table 10.1. Model Manager's reconfiguration operation signatures

Operation name	Argument name:type	Return type
replicateComponent	c : AssemblyComponent to: ExecutionContainer	DeploymentComponent
dereplicateComponent	c: DeploymentComponent	void
migrateComponent	c: DeploymentComponent to: ExecutionContainer	DeploymentComponent
allocateExecutionContainer	e: ExecutionContainer	boolean
deallocateExecutionContainer	e: ExecutionContainer	boolean

- *replicateComponent.* A deployment component, with the assembly component and the execution container given as parameters, is created in the component deployment model. A respective operation for this purpose is provided by the Component Deployment Model Manager (Section 8.2.4). The created deployment component is returned by this operation.
- *dereplicateComponent.* The passed deployment component is deleted from the component deployment model using the Component Deployment Model Manager's operation for this purpose. Note that the component is not physically removed but marked inactive because, for instance, monitoring events referencing this entity may be received after the component has been de-replicated.
- *migrateComponent.* This operation consecutively performs the steps described for the *replicateComponent* and *dereplicate* operations, returning the newly created deployment component.
- *allocateExecutionContainer.* The passed execution container is added to the execution environment model's list *allocatedExecutionContainers* of allocated execution containers (cf. Figure 6.6 on page 98). A respective operation for this purpose is provided by the Execution Environment Model Manager. In contrast to the signature of the corresponding operation added to the meta-model (page 168), the Reconfiguration Model Manager's signature expects an *ExecutionContainer* instead of an *ExecutionContainerType* as parameter. The reason is that the Reconfiguration Manager obtains a preliminary execution container—passing the type—from the Execution Environment Model Manager when start-

ing to execute the corresponding reconfiguration operations. The *allocateExecutionContainer* operation confirms this preliminary allocation.

- *deallocateExecutionContainer.* The passed execution container is removed from the execution environment model's list *allocatedExecutionContainers* of allocated execution containers. A respective operation for this purpose is provided by the Execution Environment Model Manager.

10.2.3 Reconfiguration Manager Extensions

As introduced in Section 8.5, the Reconfiguration Manager is responsible for executing architectural reconfiguration plans (Figure 6.11) by interacting with the technology-specific effector APIs. Hence, the Reconfiguration Manager is separated into a technology-agnostic (architectural) part and a technology-specific part. The technology-agnostic parts takes care of the transactional interpretation of the architectural reconfiguration plans by triggering the execution of operations from the technology-specific parts, communicating with the Model Manager for model queries and updates, including rollbacks etc.. Model queries and updates also concern the mappings between implementation-level and architecture-level entities maintained by the Arch2Technology Mapping Manager.

When adding runtime reconfiguration operations to the SLAstic framework, usually both parts of the Reconfiguration Manager need to be extended. First, the technology-agnostic parts needs to know what parts of the SLAstic runtime model need to be queried and updated as part of the specific operations. Second, technology-specific implementations for the operations need to be provided. For example, for the five SLAstic runtime reconfiguration operations, respective functionality is included in the technology-agnostic part—as detailed below—and technology-specific implementations for the Palladio Component Model (PCM) and Eucalyptus-based infrastructures exist, which will be detailed in Chapters 11 and 13.

The technology-agnostic part of the Reconfiguration Manager includes an operation (*doReconfiguration*) accepting a SLAstic *ReconfigurationPlan* (Section 6.4.1) to be executed. Remind that the data structures representing the contained runtime reconfiguration operations have been introduced in Section 10.2.1. Roughly based on the visitor design pattern [Gamma et al., 1995], the execution of runtime reconfiguration operations is dispatched to respective operation-specific handler operations. Such handler

operations must exist for each runtime reconfiguration operation. These handler operations perform the corresponding runtime model changes via the Reconfiguration Model Manager and trigger the technology-specific effector operations from the Reconfiguration Manager. Note that the execution of a plan is performed in a transactional context—(currently) in the sense that only one reconfiguration plan is executed at a time.

For the five SLAstic runtime reconfiguration operations, the technology-specific handler operations work as follows—omitting failure handling:

- `replicateComponent: a`) Call the Reconfiguration Model Manager's `replicateComponent` operation, *b*) trigger the technology-specific `replicateComponent` operation in the respective part of the Reconfiguration Manager passing the deployment component.
- `dereplicateComponent: a`) Call the Reconfiguration Model Manager's `dereplicateComponent` operation, *b*) trigger the technology-specific `dereplicateComponent` operation in the respective part of the Reconfiguration Manager passing the deployment component.
- `migrateComponent: a`) Call the Reconfiguration Model Manager's `migrateComponent` operation, *b*) trigger the technology-specific `migrateComponent` operation in the respective part of the Reconfiguration Manager.
- `allocateExecutionContainer: a`) Call the Reconfiguration Model Manager's `allocateExecutionContainer` operation, *b*) trigger the technology-specific `allocateExecutionContainer` operation in the respective part of the Reconfiguration Manager.
- `deallocateExecutionContainer: a`) Call the Reconfiguration Model Manager's `migrateComponent` operation, *b*) trigger the technology-specific `deallocateExecutionContainer` operation in the respective part of the Reconfiguration Manager.

Note that we omitted the checks of fulfilled preconditions, as enumerated in Sections 10.1 and 10.2.1, from the description. The technology-specific operations include the updates to the mapping between implementation-level and architecture-level entities maintained by the Arch2Technology Mapping Manager. As mentioned before, technology-specific implementations of the five SLAstic runtime reconfiguration will be presented in Chapters 11 and 13.

Chapter 11

Utilizing the Palladio Component Model in SLAstic

This chapter describes the integration of the Palladio Component Model (PCM) [Becker et al., 2009], introduced in Section 4.5.2, into the SLAstic approach. In SLAstic, PCM is utilized for simulation-based analysis and online performance prediction. For this purpose, we developed *a*) a M2M transformation of SLAstic models to PCM instances, called SLAstic2PCM, *b*) a decoration concept for PCM instances enabling their use as runtime models with the SLAstic framework, and *c*) the discrete-event simulator SLAstic.SIM that simulates runtime reconfigurable PCM instances and integrates with the SLAstic framework. As part of SLAstic.SIM, we extended PCM by reconfiguration support for the SLAstic operations described in Chapter 10.

This chapter is structured as follows. Section 11.1 describes the SLAstic2PCM transformation. Section 11.2 outlines the concept for decorating PCM instances by SLAstic models. Section 11.3 presents the SLAstic.SIM discrete-event simulator and its integration into the SLAstic framework. Proof-of-concept implementations for the concepts described in this chapter are available as part of the supplementary material [van Hoorn, 2014]. They are part of the implementations of the SLAstic meta-model and the SLAstic framework.

11.1 Transformation from SLAstic to PCM

In order to employ model-based performance prediction for SLAstic models, we developed the model-to-model (M2M) transformation (Section 2.2) called SLAstic2PCM. From an instance of the SLAstic meta-model, SLAstic2PCM

Table 11.1. High-level mapping between SLAstic and PCM meta-model partitions

SLAstic	PCM
—	(Resource Repository Model)
Type Repository Model, Usage Model	Repository Model
Component Assembly Model	System Model
Execution Environment Model, Type Repository Model	Resource Environment Model
Component Deployment Model	Allocation Model
Usage Model	Usage Model

generates a PCM instance that can be used with existing Palladio tools, including Palladio-Bench, transformations from PCM to other performance models, as well as our SLAstic.SIM simulator for runtime reconfigurable PCM models.

The transformation results in a basic—but complete—PCM instance, which needs further refinement and calibration, e. g., w.r.t. resource demands in RDSEFFs. The reason is that the performance modeling features provided by the SLAstic meta-model are by far not as expressive as those provided by the PCM. However, this manual refinement is explicitly supported in the SLAstic framework, for example, by the SLAstic-PCM model decoration concept described in Section 11.2.

This section provides a textual and language-independent description of the transformation. The description refers to elements of the PCM and SLAstic meta-models. For details on these meta-models, please refer to Becker et al. [2009] and Chapter 6 respectively, as well as the Ecore-based implementations of the meta-models that are also available via the supplementary material [van Hoorn, 2014]. We will use the Bookstore system, already known from previous chapters, as a running example to demonstrate different steps of the SLAstic2PCM transformation. An ATL-based (see page 19) SLAstic2PCM implementation has been developed and will be presented briefly. Note that considerable parts of the research presented in this section were conducted in the context of the Diploma thesis by Günther [2011], as mentioned in Section 5.3.1. This section includes contents from this work.

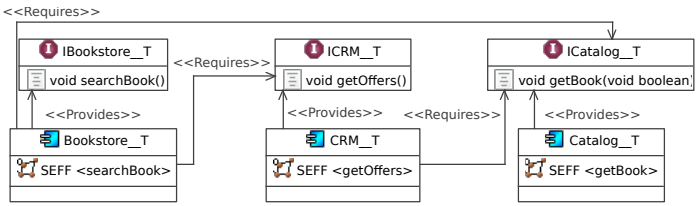


Figure 11.1. Diagram of the PCM repository for the Bookstore created by SLAstic2PCM. An RDSEFF for the *searchBook* operation is shown in Figure 11.2.

Table 11.1 provides a first high-level of the relationship between the partitions of the SLAstic and the PCM meta-model. Sections 11.1.2 to 11.1.6 describe the generation for each of the PCM partitions, i. e., repository, system, resource environment, allocation, and usage. Section 11.1.7 briefly describes the implementation of SLAstic2PCM, which employs the ATLAS Transformation Language (ATL). Section 11.1.8 lists current limitations.

11.1.1 Generation of the PCM Resource Repository

As depicted in Table 11.1, the generation of the PCM resource repository (*pcm::ResourceRepository*[1]) does not rely on any SLAstic model partition. It simply includes the creation of the PCM resource types for CPU, HDD, LAN, and delay resources.

11.1.2 Generation of the PCM Repository Model

A PCM repository (*pcm::Repository*) is created, assigning constant values to its *id*, *entityName*, and *repositoryDescription* attributes.[2] The lists of contained interfaces and components, i. e., *interfaces* and *components*, are obtained by transforming each element of the corresponding lists contained in the *slastic::TypeRepositoryModel*, i. e., *interfaces* and *componentTypes*. Additional

[1] In this section, meta-classes from the PCM meta-model are prefixed by *pcm::*. Accordingly, SLAstic meta-classes are prefixed by *slastic::*. Package names are omitted to improve readability.

[2] In the PCM meta-model, attribute names include the name of the owning class as a name postfix, e. g., *childComponentContexts_ComposedStructure*. We omit this postfix unless needed.

information from the *slastic::UsageModel* is used to create Resource Demanding SEFFs (RDSEFFs) contained in the *pcm::Repository's* components. The remaining parts of this section detail these three core repository-related generation steps. Figure 11.1 shows a diagram for the PCM repository, which results from the application of SLAstic2PCM to the SLAstic instance of the Bookstore.

Interfaces

Each *slastic::Interface*[3] is transformed into a *pcm::Interface*. The values of the *slastic::Interface's* attributes *id* and *name* are assigned to the *pcm::Interface's* attributes *id* (with prefix "i")[4] and *entityName*.

The contained list of *pcm::Signatures* (*signatures*) is generated by transforming the *slastic::Interface's* corresponding list of *slastic::Signatures* (*signatures*). For each *pcm::Signature*, the *serviceName* is assigned the value of the corresponding *slastic::Signature's name*; the list of *pcm::Parameters* (*parameters*) is generated from the *slastic::Signature's* list of parameter types (*paramTypes*) .

Component Types

Each *slastic::ComponentType* is transformed into a *pcm::BasicComponent*. The PCM meta-model supports the composition of *pcm::BasicComponents* into *pcm::CompositeComponents*. The *slastic::ComponentType's* attribute values for *id* and *name* are assigned to the *pcm::BasicComponent's* attributes *id* (prefix "c") and *entityName*.

The *pcm::BasicComponent's* list of *pcm::ProvidedRoles* (*providedRoles*) is obtained by creating a *pcm::ProvidedRole* for each *slastic::Interface* contained in the *slastic::ComponentType's providedInterfaces* list. The *id* and *entityName* of the each *pcm::ProvidedRole* is created by concatenating the *ids*/*names* of the *slastic::Interface* and *slastic:ComponentType*, prefixed by "pr" and "Provided_" respectively. For example, for a *slastic::Interface* with *name* "ICRM_T" and a *slastic:ComponentType* with *name* "CRM_T", this results in an

[3]We use meta-class names as words in sentences. Plural and possessive forms of the meta-class are used to improve readability, even though the actual meta-class name is singular.

[4]Because identifiers in the SLAstic meta-model are unique among entity types, prefixes are used to make identifiers unique across the resulting PCM model.

entityName "Provided_ICRM_T_CRM_T". The *pcm::ProvidedRole's providedInterface* references the matching *pcm::Interface*, obtained by the above-described interface transformation.

Accordingly, the *pcm::BasicComponent's* list of *pcm::RequiredRoles* (*requiredRoles_InterfaceProvidingEntity*) is obtained by creating a *pcm::RequiredRole* for each *slastic::Interface* in the *slastic::ComponentType's requiredInterfaces* list.

RDSEFFs

Any *pcm::BasicComponent* contains a list of *pcm::ServiceEffectSpecifications*. Each *pcm::ServiceEffectSpecification* describes the implementation of a *pcm::Signature*. In order to to be valid, a *pcm::BasicComponent* must contain a *pcm::ServiceEffectSpecification* for each *pcm::Signature* declared by the *pcm::ProvidedRole's pcm::Interfaces*. A concrete implementation, extending *pcm::ServiceEffectSpecification*, is *pcm::ResourceDemandingSEFF*. As a *pcm::ResourceDemandingBehavior*, it contains a list of *pcm::AbstractActions*, whose first and last elements must be a *pcm::StartAction* and *pcm::StopAction* respectively. A *pcm::ExternalCallAction* models the call to an external service specified by its *pcm::Role* and *pcm::Signature*. A *pcm::InternalAction* quantifies (*pcm::PCMRandomVariable*) a *pcm::ParametricResourceDemand* to a referenced *pcm::ProcessingResourceType*. *pcm::LoopAction* and *pcm::BranchAction* are available to model loops and branches, which both again contain a list of *pcm::AbstractActions*. A number of additional *pcm::AbstractActions* exist.

A *pcm::ResourceDemandingSEFF* is created from a *slastic::Operation* and the corresponding *slastic::OperationCallFrequency* and *slastic::CallingRelationship* information contained in the *slastic::UsageModel*. The created *pcm::ResourceDemandingSEFF* references the *pcm::Signature* (*describedService*) that corresponds to the *slastic::Operation's slastic::Signature*.

For each *pcm::ResourceDemandingSEFF*, SLAstic2PCM creates the behavioral specification in terms of the *pcm::AbstractAction* according to the following pattern (see also [Günther, 2011]). Figure 11.2 shows a generated RDSEFF of the Bookstore's *searchBook* method, included in the PCM repository in Figure 11.1.

1. Creation of a *pcm::StartAction*.
2. Creation of a *pcm::InternalAction* with a *pcm::ParametricResourceDemand*

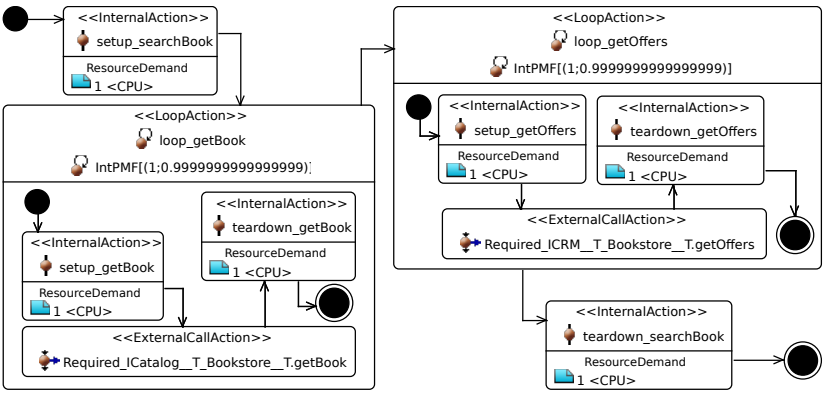


Figure 11.2. Diagram for an RDSEFF of the Bookstore's *searchBook* operation created by SLAstic2PCM

(*resourceDemand*) to the *pcm::ResourceType* 'CPU'. The specification of the demand is set to a fixed value. This *pcm::InternalAction* models a computation executed after entering a software operation.

3. Creation of a sequence of *pcm::LoopActions*—one for each *slastic::CallingRelationship* referencing the current *slastic::Operation* as its calling operation (*callingOperation*). Such a *pcm::LoopAction* for a *slastic::CallingRelationship* is created as follows:

   - The *pcm::ResourceDemandingBehavior* (*bodyBehavior*) consists of a sequence of the following *pcm::AbstractActions*: *pcm::StartAction*, *pcm::InternalAction*, *pcm::ExternalCallAction*, *pcm::InternalAction*, and *pcm::StopAction*. The two *pcm::InternalActions*, again with a (fix) *pcm::ParametricResourceDemand* to a 'CPU' *pcm::ResourceType*, model setup and tear down computations that are executed before and after the *pcm::ExternalCallAction*. The *pcm::ExternalCallAction's pcm::Signature calledService* and *pcm::Role* (*role*) are assigned the corresponding entities created from the *slastic::Operation* (*callingOperation*), *slastic::Interface* (*calledInterface*), and *slastic::Signature* (*calledSignature*), which are referenced by the *slastic::CallingRelationship*.

- The *pcm::LoopAction's* iteration count (*iterationCount* of type *pcm::PCMRandomVariable*) is created from the *slastic::CallingRelationship's slastic::FrequencyDistribution*.

  Let $c_{1...n}$, with $c_i \in \mathbb{N}^+$, the sorted set of invocation counts to a *slastic::Interface's slastic::Signature* and $f_{1...n}$, with $f_i \in \mathbb{N}^+$, the corresponding list of absolute frequencies for these counts, which are both contained in the *slastic::CallingRelationship's slastic::FrequencyDistribution*, as sequences *values* and *frequencies*. The absolute number s of *slastic::Operation* executions that are involved in one or more calls to the *slastic::Interface's slastic::Signature* is the sum of absolute frequencies:

  $$s = \sum_{i=1}^{n} f_i \tag{11.1}$$

  Using s, we can compute the relative frequencies $r_{1...n}$ from the absolute frequencies $f_{1...n}$:

  $$r_i = \frac{f_i}{s} \tag{11.2}$$

  A stochastic expression for the iteration count is created to be used as the *pcm::PCMRandomVariable's specification*. In this case, the created stochastic expression is a probability mass function given in PCM's stochastic expression notation:

  $$IntPMF[(c_1;\ r_1) \cdots (c_i;\ r_i) \cdots (c_n, r_n)] \tag{11.3}$$

4. Creation of a *pcm::InternalAction* with a *pcm::ParametricResourceDemand* (*resourceDemand*) to the *pcm::ResourceType* 'CPU'. Again, the specification of the resource demand is set to a fixed value. This *pcm::InternalAction* models a computation executed before exiting a software operation.

5. Finally, a *pcm::StopAction* is created.

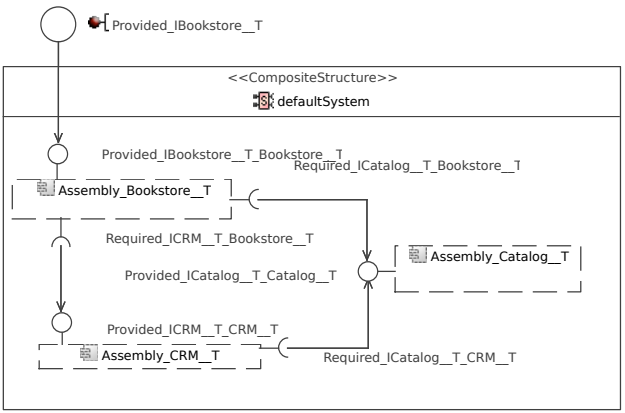


Figure 11.3. Diagram of the PCM system model for the Bookstore created by SLAstic2PCM

11.1.3 Generation of the PCM System Model

In PCM, a *pcm::System* is a *pcm::ComposedProvidingRequiringEntity*, which is also used to model *pcm::CompositeComponents* (see Section 11.1.2). A *pcm::ComposedProvidingRequiringEntity* is a *pcm::InterfaceProvidingRequiringEntity*, as described in Section 11.1.2, and a *pcm::ComposedStructure*. A *pcm::ComposedStructure* contains a set of *pcm::AssemblyContexts*, which are basically instances of a component type defined in the *pcm::Repository* (Section 11.1.2). Three different kinds of *pcm::Connectors* are used to connect *pcm::AssemblyContexts* contained in a *pcm::ComposedStructure*—*a*) *pcm::ProvidedDelegationConnectors* and *b*) *pcm::RequiredDelegationConnectors* connect matching *pcm::ProvidedRoles* (*pcm::RequiredRoles* respectively) of a *pcm::ComposedStructure* and a *pcm::AssemblyContext*; *c*) a *pcm::AssemblyConnector* connects a *pcm::RequiredRole* of a *pcm::AssemblyContext* to the *pcm::ProvidedRole* of a second *pcm::AssemblyContext*.

A *pcm::System* is created from a *slastic::ComponentAssemblyModel*. Constant values are assigned to the attributes *id* and *entityName*. The contained *AssemblyContexts* (*childComponentContexts*) and *pcm::AssemblyConnectors* (*compositeAssemblyConnectors*) are created from the *slastic::ComponentAssemblyModel's* lists of *slastic::AssemblyComponents* (*assemblyComponents*) and

slastic::AssemblyComponentConnectors (*assemblyComponentConnectors*). The *slastic::ComponentAssemblyModel's* list of *slastic:SystemProvidedInterfaceDelegationConnectors* (*systemProvidedInterfaceDelegationConnectors*) is used to generate both the *pcm::System's* lists of *pcm::ProvidedRoles* (*providedRoles*) and *pcm:ProvidedDelegationConnector* (*providedDelegationConnectors*). These transformations are detailed in the remainder of this section.

Figure 11.3 shows a diagram for the PCM system model, which results from the application of SLAstic2PCM to the SLAstic instance of the Bookstore.

Components and Connectors

The generation of a *pcm::AssemblyContext* from a *slastic::AssemblyComponent* is straightforward: *id* and *entityName* are assigned the *slastic::AssemblyComponent's id* (prefixed by `"ac"`) and *name* (prefixed by `"Assembly_"`); *encapsulatedComponent* references the *pcm::BasicComponent* corresponding to the *slastic::AssemblyComponent's slastic::ComponentType*.

A *pcm::AssemblyConnector* is created from a *slastic::AssemblyComponentConnector*. It references the *pcm::AssemblyContexts* (*requiringChildComponentContext* and *providingChildComponentContext*), obtained by the transformation described in the previous paragraph. Additionally, it references the matching *pcm::RequiredRole* (*requiredRole*) and the *pcm::ProvidedRole* (*providedRole*), defined for the *pcm::AssemblyContext's* encapsulated type (Section 11.1.2). The *entityName* is assigned the value of the *slastic::AssemblyComponentConnector's name* attribute.

Roles and Delegation Connectors

Each *pcm::ProvidedRole* is generated based on the *slastic:SystemProvidedInterfaceDelegationConnector's slastic::Interface*: *id* and *entityName* are set to the *slastic::Interface's id* (prefixed by `"p"`) and *entityName* (prefixed by `"Provided_"`). The *providedInterface* is the *pcm::Interface* corresponding to this *slastic::Interface*.

A *pcm::ProvidedDelegationConnector* is created for each *slastic::SystemProvidedInterfaceDelegationConnector*. The attribute *childComponentContext* references the *pcm:AssemblyContext* corresponding to the *slastic::AssemblyComponent* referenced by the *slastic::SystemProvidedInterfaceDelegationConnector*.

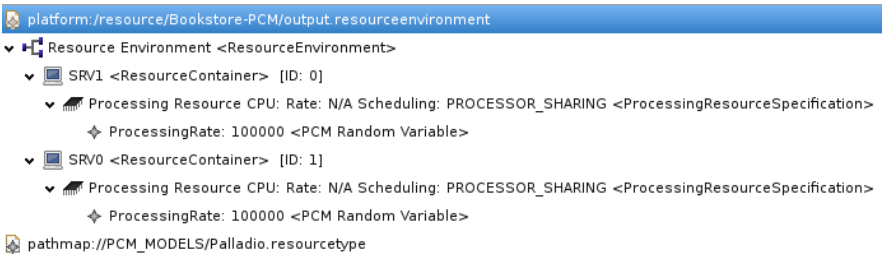


Figure 11.4. PCM resource environment model created by SLAstic2PCM (Bookstore system)

The *innerProvidedRole* references matching *pcm::ProvidedRole* of the *pcm:AssemblyContext's* type (Section 11.1.2).

Accordingly, the *pcm::System's* lists of *pcm::RequiredRoles* (*requiredRoles*) and *pcm:RequiredDelegationConnector* (*requiredDelegationConnectors*) are generated from the *slastic::ComponentAssemblyModel's* list of *slastic:SystemRequiredInterfaceDelegationConnectors*, i. e., *systemRequiredInterfaceDelegationConnectors*.

11.1.4 Generation of the PCM Resource Environment Model

The *pcm::ResourceEnvironment* contains the lists of *pcm::ResourceContainers* (*resourceContainer*) and *pcm::LinkingResources* (*linkingresource*). Each *pcm::ResourceContainer* includes a list of *pcm::ProcessingResourceSpecifications*, each being an instance of a *pcm::ProcessingResourceType* (i. e., CPU, hard disk, or delay) from the *pcm::ResourceRepository* along with an associated *pcm::SchedulingPolicy* and a processing rate (*pcm::PCMRandomVariable*). A *pcm::LinkingResource* with a given *pcm::CommunicationLinkResourceSpecification* connects two *pcm::ResourceContainers*. The *pcm::CommunicationLinkResourceSpecification* specifies the *pcm::CommunicationLinkResourceType* along with the latency and throughput (*pcm::PCMRandomVariable*).

A *pcm::ResourceEnvironment* is created from a *slastic::ExecutionEnvironmentModel*. The *pcm::ResourceEnvironment's* list of *ResourceContainers* (*resourceContainer*) is created from the *slastic::ExecutionEnvironmentModel's* list of *slastic::ExecutionContainers* (*executionContainers*). The values of the *pcm::-*

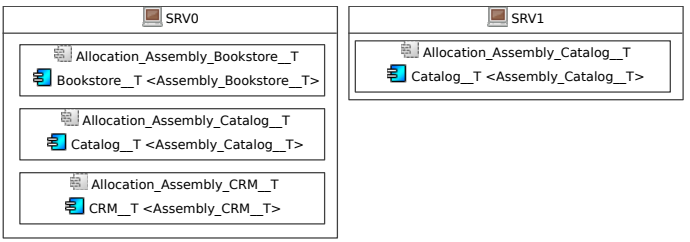


Figure 11.5. Diagram of the PCM allocation model created by SLAstic2PCM

ResourceContainer's id and *entityName* attributes are assigned the values of the *slastic::ExecutionContainer's* attribute values *id* and *name*. The *pcm::ResourceContainer's* list of *ProcessingResourceSpecifications* (*activeResourceSpecifications*) is obtained by transforming the *slastic::ExecutionContainer's* list of *slastic::ResourceSpecifications* (*resources*).

Figure 11.4 shows the PCM resource environment model, which results from the application of SLAstic2PCM to the SLAstic instance of the Bookstore.

11.1.5 Generation of the PCM Allocation Model

The *pcm::Allocation* contains a list of *pcm::AllocationContexts*. Each *pcm::AllocationContext* references a *pcm::AssemblyContext* and a *pcm::ResourceContainer*.

A *pcm::Allocation* is created from a *slastic::ComponentDeploymentModel*. Its *id* attribute is set to a fixed value and it references the *pcm::ResourceEnvironment* (*targetResourceEnvironment*) and the *pcm::System* (*system*), both created by this transformation (Sections 11.1.3 and 11.1.4). The *pcm::Allocation's id* is set to a fixed value. The list of *pcm::AllocationContexts* (*allocationContexts*) is obtained by transforming each element in the *slastic::ComponentDeploymentModel's* list of *slastic::DeploymentComponents*. A *pcm::AllocationContext* is created from a *slastic::DeploymentComponent* by assigning the *pcm::AssemblyContext* (*assemblyContext*) and the *pcm::ResourceContainer* (*resourceContainer*), which correspond to the *slastic::AssemblyComponent* (*assemblyComponent*) and the *slastic::ExecutionContainer* (*executionContainer*) referenced by the *slastic::DeploymentComponent* and which are created by the transformations de-

scribed in Sections 11.1.3 and 11.1.4. The *pcm::AllocationContext's id* and *entityName* are assigned the values of the *slastic::DeploymentComponent's id* (prefixed by "alc") and, respectively, the *entityName* (prefixed by "Allocation_") of the referenced *pcm::AllocationContext*.

Figure 11.5 shows a diagram for the PCM allocation model, which resulted from the application of SLAstic2PCM to the SLAstic instance of the Bookstore.[5]

11.1.6 Generation of the PCM Usage Model

A *pcm::UsageModel* contains a list of *pcm::UsageScenarios*, each of which contains a specification of the *pcm::Workload* and the *pcm::ScenarioBehavior*. The *pcm::Workload* can either be a *pcm::OpenWorkload*, with a given inter-arrival time (*pcm::PCMRandomVariable*), or a *pcm::ClosedWorkload* with a given population size and think time (*pcm::PCMRandomVariable*). A *pcm::ScenarioBehavior* contains a sequence of *pcm::AbstractUserActions*. Basic *pcm::AbstractUserActions* are *pcm::Delay* and *pcm::EntryLevelSystemCall*. A *pcm::EntryLevelSystemCall* models the invocation of a service provided by a *pcm::System*, referenced by the corresponding *pcm::ProvidedRole* and *pcm::Signature*. The *pcm::AbstractUserActions pcm::Loop* and *pcm::Branch* contain nested *pcm::ScenarioBehaviors*, with associated probabilistic (*pcm::PCMRandomVariable*) loop iteration counts and branch probabilities respectively. Each (sub)sequence of *pcm::AbstractUserActions* starts and stops with *pcm::Start* and *pcm::Stop* respectively.

A *pcm::UsageModel* is created from the list of *slastic:SystemProvidedInterfaceDelegationConnectors* (*systemProvidedInterfaceDelegationConnectors*) and the associated information contained in the *slastic::UsageModel*. For each *slastic::SystemProvidedInterfaceDelegationConnector*, a *pcm::UsageScenario* with a *pcm::OpenWorkload* specification is created. The *pcm::ScenarioBehavior* (*scenarioBehavior*) contained in the *pcm::UsageScenario* consists only (in addition to *pcm::Start* and *pcm::Stop*, of course) of a single *pcm::EntryLevelSystemCall* referring to the *pcm::Signature* and the *pcm::System's ProvidedRole* created for the *slastic::SystemProvidedInterfaceDelegationConnector* (Section 11.1.3).

[5]Note that *this* configuration cannot be executed in the Palladio-Bench, because each *pcm::AssemblyContext* must be allocated exactly once—in this case, *Catalog* is allocated twice.

Figure 11.6 shows a diagram for a `pcm::UsageScenario` contained in the `slastic::UsageModel`, which resulted from the application of SLAstic2PCM to the SLAstic instance of the Bookstore.

11.1.7 Implementation of SLAstic2PCM

SLAstic2PCM was implemented with the ATLAS Transformation Language (ATL), introduced in Section 2.3.2. As depicted in Figure 11.7, the ATL module `slastic2pcm.atl` creates a PCM instance, consisting of six PCM submodels, from a SLAstic system model and a SLAstic usage model. Figure 11.8 lists an example transformation rule from the ATL-based SLAstic2PCM transformation. The complete ATL module is part of the SLAstic framework, available via the supplementary material [van Hoorn, 2014]. It has a length of 518 lines and includes 37 transformation rules. Additional details on SLAstic2PCM's implementation are described by Günther [2011].

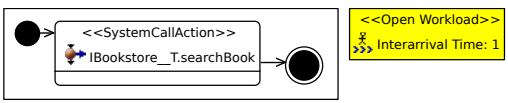


Figure 11.6. Diagram for the `pcm::UsageScenario` created by SLAstic2PCM

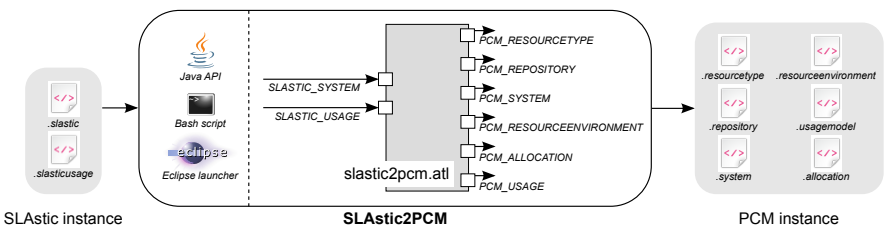


Figure 11.7. Implementation of the SLAstic transformation SLAstic2PCM. The ATL-based transformation can be invoked via *a*) Java API, *b*) Bash script, and *c*) an Eclipse launch configuration.

```
rule ExecutionContainerToResourceContainer {
    from src: Slastic!ExecutionContainer
    to tgt: Pcm!ResourceContainer in RESOURCEENVIRONMENT (
        id <- src.id.toString(),
        entityName <- src.name,
        activeResourceSpecifications_ResourceContainer
                <- src.executionContainerType.resources
    )
}
```

Figure 11.8. Example SLAstic2PCM rule for transforming a SLAstic execution container into a PCM resource container

Three mechanisms are provided to invoke the ATL-based SLAstic2PCM implementation, namely via Java API, via Bash script and via an Eclipse launch configuration (Figure 11.7). The Java API allows to invoke the transformation within Java programs, e. g., as part of the SLAstic framework. The Bash script wraps a *main* method included in the afore-mentioned Java API. Finally, the Eclipse launch configuration, provided by the ATL tool infrastructure, can be used to invoke the transformation inside the Eclipse IDE.

11.1.8 Current Limitations

The current limitations of the SLAstic2PCM transformation can be divided into three categories: *a*) concepts of the PCM meta-model not being generated, *b*) missing parts of transformations on the conceptual level, and *c*) parts of the transformation not included in the prototype implementation. The remainder of this section lists the limitations based on this categorization, merging the latter two categories. Note that we expect that each of the critical limitations can be resolved with little effort as both all required information is available and the implementation is straightforward.

The PCM instances created by the transformation do not include all of the concepts provided by the PCM meta-model. Particularly, *slastic::Com-*

ponentTypes are transformed into *pcm::BasicComponents* instead of using *pcm::CompositeComponents*. The reason is that information about component (type) composition is not needed for the SLAstic approach so far, and as such, is not included in the meta-model. Hence, the decision to transform to *pcm::BasicComponents* is straightforward. However, *pcm::BasicComponents* may be detailed as *pcm::CompositeComponents* as part of the manual refinement step. Other concepts currently not being generated by the transformation include *pcm::PassiveResources*, and additional types of *pcm::Actions* as part of RDSEFFs (e. g., w.r.t. asynchronous behavior).

Limitations with respect to the transformation on a conceptual and implementation level comprise the following:

- As part of the transformation from *slastic::Signatures* to *pcm::Signatures*, *pcm::DataTypes* and *pcm::ExceptionTypes* are not created from the type information in the SLAstic model and assigned to the respective attributes. These types are currently modeled as strings in the SLAstic meta-model because additional information is not needed.

- As part the *pcm::System* generation (Section 11.1.3), the creation of the system's required roles and corresponding delegation connectors are currently not included in the implementation. However, this information is available in the SLAstic model.

- As part of the *pcm::ResourceEnvironment* generation (Section 11.1.4), the creation of *pcm::LinkingResources* and their connection with *pcm::ResourceContainers* is currently not implemented. However, this information is available in the SLAstic model. The *processingRate* is currently set to a fixed value (100), but, e. g., the *slastic::CPUType's clockRateMhz* could be used. The *pcm::SchedulingPolicy* is not set because this information is not available in the SLAstic meta-model. Processor sharing is used as default.

- As part of the *pcm::UsageModel* (Section 11.1.6), the *pcm::OpenWorkload*'s *interArrivalTime* is currently set to a fixed value (1) rather than exploiting the information contained in the *slastic::UsageModel*, i. e., frequency and observation time.

- The transformation is based on an old version of the PCM meta-model.

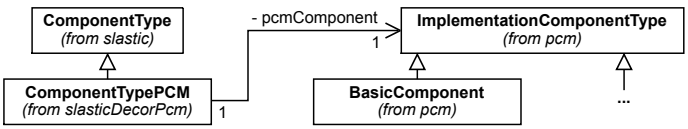


Figure 11.9. PCM decoration example for component types

11.2 Decoration of PCM Instances

The focus of the SLAstic meta-model is to serve as an architectural runtime model for architecture-based online capacity management in the SLAstic framework. As opposed to the SLAstic meta-model, PCM aims to be a full-blown modeling language for model-based performance prediction. Hence, it needs to include a comprehensive set of performance-relevant details. However, as described in Chapter 8, online performance prediction is one of the activities of proactive online capacity management. In order to benefit from the performance prediction capabilities provided by the PCM infrastructure, we developed a basic concept to link SLAstic runtime models with a corresponding PCM instance.

The basic approach is that extending meta-classes for a subset of the SLAstic meta-classes exist, which include a model reference to the corresponding entity from the PCM meta-model. As an example, Figure 11.9 depicts the decoration of a PCM component type by a SLAstic component type. In this case, `ComponentTypePCM` serves as the decoration class. Based on this pattern, other meta-classes exist that extend SLAstic's meta-classes and serve to decorate respective PCM classes.

Two particular use cases for the decoration exist. First, assume an existing SLAstic model, which is transformed into a PCM instance—e. g., employing the SLAstic2PCM transformation described in Section 11.1. In this case, the resulting PCM may be refined in the PCM tooling infrastructure, while retaining the connection to the source SLAstic model. At runtime, the connected PCM instance can be used for performance prediction. Second, a PCM instance may exist that is to be used at runtime in the SLAstic framework. In this case, a respective SLAstic model employing the decorator meta-classes is generated. Note that such transformation does not exist, yet. In both cases, we refer to this as *decoration*, as for components

working with the SLAstic runtime model, the connection to the PCM model is transparent, i. e., not visible. However, note that the Model Manager needs to be aware of this connection as changes to the SLAstic model need to be propagated to the PCM instance.

Both use cases provide an argument why we chose to use `pcm::ImplementationComponentType` as the reference type of the component type decorator meta-class depicted in Figure 11.9. As detailed in Section 11.1, a `slastic::ComponentType` is transformed into a `pcm::BasicComponent`. However, in PCM other component types exist, e. g., `pcm::CompositeComponent`. Such types enable more detailed modeling, e. g., w.r.t. a type's internal structure. Having chosen `pcm::ImplementationComponentType` as the reference type allows to retain the connection even after a refactoring action to a generated model or to a PCM instance, which is to be decorated and includes such construct.

11.3 Simulation of Runtime Reconfigurable PCM Instances

This section describes the SLAstic.SIM discrete-event simulator for runtime reconfigurable PCM models and its integration into the SLAstic framework. Considerable parts of the research presented in this section were conducted in the context of the Diploma thesis by von Massow [2010] (as mentioned in Section 5.3.1) and our joint publication on this work [von Massow et al., 2011]. This section includes contents from these two publications, which may serve as a reference for further details on SLAstic.SIM. For an introduction into simulation, please refer to respective textbooks, e. g., by Banks [1998], Banks et al. [2009], and Page and Kreutzer [2005].

The remainder of this section is structured as follows. Sections 11.3.1 and 11.3.2 give an overview of SLAstic.SIM as as well as its architecture and integration into the SLAstic framework. In Section 11.3.3, we describe how the SLAstic reconfiguration operations from Chapter 10 are implemented within SLAstic.SIM using PCM. The execution of the simulation model is described in Section 11.3.4.

11.3.1 Overview

Like the SimuCom reference simulator, which is included in the Palladio-Bench, SLAstic.SIM simulates PCM models in order to obtain performance-relevant properties from the simulated system. However, as opposed to SimuCom, SLAstic.SIM can be considered an adaptive software system as introduced in Section 3.4.1 as it provides an effector and sensor API like a real software system. Like SLAstic.SIM, SimuCom is implemented employing Desmo-J. SimuCom's simulation code is completely generated from a PCM instance employing a M2T transformation prior to simulation start. This approach is well-suited for software architectures, which are not reconfigured during simulation. However, it is not trivial to extend SimuCom's M2T transformation by simulation support of runtime reconfigurable PCM instances. This was one of the main reasons for us to develop a new simulator for PCM models with runtime reconfiguration support, following an interpretive simulation approach. Another reason was that the SimuCom simulations are only executable in an OSGi [OSGi Alliance, 2012] environment like Eclipse. As opposed to SimuCom, SLAstic.SIM currently does not support the simulation of middleware models.

SLAstic.SIM has been developed for two use cases within the SLAstic approach: simulative evaluation and online performance prediction. The first use case has already been outlined in Section 8.6.2 (illustrated in Figure 8.6c): SLAstic.SIM is employed to replace a real system connected to the SLAstic framework. SLAstic.SIM simulates an adaptable component-based software system, continuously provides monitoring data to the SLAstic framework, and receives reconfiguration triggers to be incorporated into the further simulation. The second use case refers to the use of SLAstic.SIM as part of the performance prediction and adaptation planning activities mentioned in Section 8.4.3. Included in SLAstic.SIM is a PCM-specific implementation of the SLAstic runtime reconfiguration operation described in Chapter 10. Integrated into SLAstic.SIM is Kieker for logging and providing (simulated) monitoring data, as well as for receiving workload events.

11.3.2 Architecture and Framework Integration

SLAstic.SIM's core components, relationships, and its integration into the SLAstic framework are depicted in Figure 11.10. For the SLAstic.Control

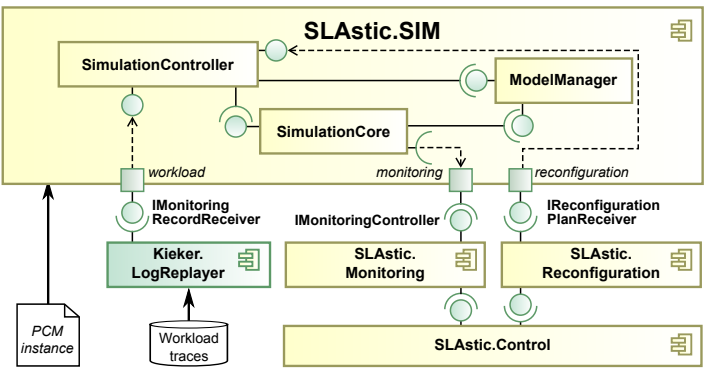


Figure 11.10. High-level architecture and integration of SLAstic.SIM [von Massow et al., 2011] (cf. Figure 8.6c).

component, SLAstic.SIM emulates a real software system with runtime reconfiguration capabilities. The `SimulationController` is responsible for the simulation life-cycle and for handling external events. Initially, the input PCM instance is transformed into an internal representation used during simulation and maintained by the `ModelManager`. The `ModelManager` includes a `ReconfigurationController` and a controller for each PCM model partition, e. g., an `AllocationController`. The `SimulationCore` executes the simulation including the generation and execution of internal simulation events. SLAstic.SIM employs the Java-based discrete-event simulation framework Desmo-J [Page and Kreutzer, 2005]. Communication with SLAstic.SIM is possible via the `workload`, `monitoring`, and `reconfiguration` ports. These ports allow to *a*) input the workload driving the simulation, *b*) receive the performance data generated during simulation, and *c*) request reconfigurations to be executed by the simulator, as detailed below. Our monitoring and analysis framework Kieker is used for reading the workload traces and for logging the simulation data. The remainder of this section provides additional details on external workload traces, monitoring, and reconfiguration.

Workload

Workload is received from a `Kieker.LogReplayer` component which reads workload traces from a monitoring log and passes them to registered plugins (Section 7.1.3)—in this case SLAstic.SIM. As these logs typically contain complete control flow traces and not just the top-level entry calls, the `SimulationController` filters the incoming workload and delegates it to the `SimulationCore`.

Monitoring

Currently, SLAstic.SIM includes probes for collecting the following information during simulation:

- *Executions.* Each simulated execution of external calls is monitored with the associated information on the service and assembly context, the resource container, the entry and exit times, as well as the control flow information. *OperationExecutionRecords* (Section 7.2.2) are used to represent this data.
- *CPU utilization.* For each CPU of allocated resource containers, the utilization is measured in intervals of 0.5 simulated time units. *CPUUtilizationRecords* (Figure 7.2) are used to represent this data.
- *Active users.* If a call from outside of the system occurs, we increment the user count and write a monitoring record. Upon the return of a call the user count is decremented again and another record is written. We defined a custom monitoring record type to represent this data.

The monitoring records are created by the probes and passed to the SLAstic.Monitoring component via Kieker's `MonitoringController`. The monitoring probes are injected using the Google Guice[6] dependency injection framework. This gives the possibility to enable or disable probes between different simulation runs by simply replacing a class's implementation. It is also possible to disable each of these probes separately or to add additional ones.

[6]Google Guice: `http://code.google.com/p/google-guice/`

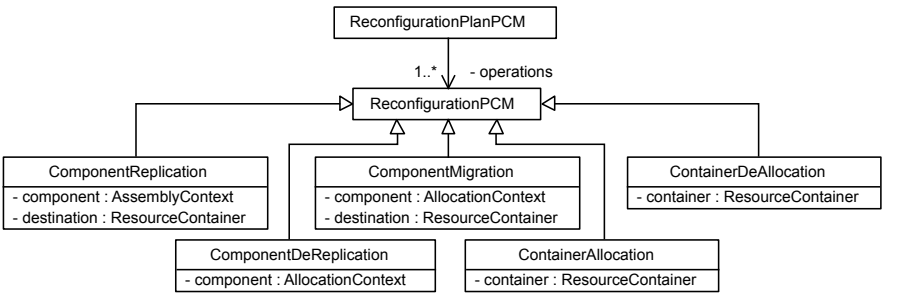


Figure 11.11. SLAstic.SIM reconfiguration plan and operations [von Massow et al., 2011]

Reconfiguration

The `IReconfigurationPlanReceiver` interface (Figure 11.10) makes it possible to send reconfiguration plans to the simulator. Plans are received and checked by the `SimulationController`, which forwards it to the `ModelManager`. The `ModelManager` translates each plan into internal reconfiguration events. These events are simulated by the `SimulationCore`. As depicted in Figure 11.11, a plan includes a list of reconfiguration actions to be executed, which is similar to the respective part of the SLAstic meta-model (Section 6.4.1). These action are successively applied to the simulation model by the `ReconfigurationController`. Each operation of a reconfiguration plan is transformed into one or more events, which are scheduled and executed consecutively. If an event fails to execute, the current plan is aborted. The following section details the PCM-specific implementation and execution of the SLAstic runtime reconfiguration operations from Chapter 10 (cf. Figure 11.11).

11.3.3 PCM-Specific Runtime Reconfiguration Operations

Figure 11.11 includes the signatures of the PCM-specific implementations of the five SLAstic runtime reconfiguration operations for controlling capacity, described in Chapter 10. The execution of these operations will be described in the remainder of this section. Additional details are provided by von Massow [2010].

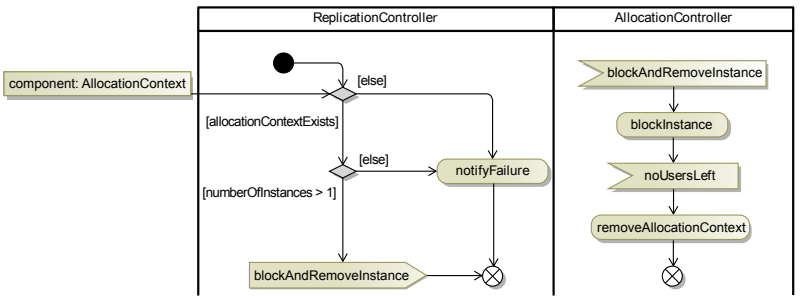


Figure 11.12. Activity diagram for the component de-replication operation [von Massow, 2010]

1. *Component replication.* For the given assembly context, a new allocation context located on the destination container is created and added to the model. The destination container must be allocated prior to the call and must not contain an allocation context for this assembly context.
2. *Component de-replication.* The existing allocation context is blocked, which means that no new calls are dispatched to this instance. As soon as all running transactions handled by the component are finished, the allocation context is removed from the model. Prior to the request, at least two allocation contexts must exist for the assembly context. The activity diagram in Figure 11.12 depicts the execution of a de-replication operation within SLAstic.SIM.
3. *Component migration.* The migration is implemented by executing a replication followed by a de-replication operation. Hence, the new allocation context immediately handles new calls while the old allocation context exists until all executing calls are finished.
4. *Container de-allocation.* The container is marked unavailable which means that it cannot be the target of migration or replication operations until it is allocated again. Prior to the request, the resource container to be de-allocated must be allocated and empty—i. e., it must not contain any allocation context.
5. *Container allocation.* The container is marked available which means that it can be the target of migration or replication operations. The operation

can be executed if the resource container exists in the simulation model and is not allocated at that time. Upon completion, components can be replicated or migrated to it. Initially, exactly those resource containers from the resource environment being associated with at least one deployment context are marked as allocated.

11.3.4 Simulation

Desmo-J offers two styles of modeling [Page and Kreutzer, 2005]: process-based and event-based. We chose to use the event-based model as all our state changes in the simulation model are instantaneous and there would be no real life-cycle. Below, we give a brief overview of the generation and execution of control flows.

Control Flow Generation

On each external call from the input workload, the complete control flow chain is generated. This is done by traversing and evaluating the corresponding RDSEFF. Call enter and return events are generated for each *ExternalCallAction*.[7] For each *InternalAction*, an internal action event is produced, containing the resource demands of the input *InternalAction*. *BranchActions* are evaluated by deciding which transition to take and traversing the transition's body. *LoopActions* are evaluated similarly by determining the iterations and then traversing the body for each iteration. The result is a list of Desmo-J events which are scheduled consecutively. Note that SLAstic.SIM's support for parametric stochastic expressions, e. g., as part of parametric resource demand specifications of *InternalActions* as well as iteration counts (*LoopAction*) and branch conditions (*BranchAction*), is currently limited to constants.

Execution of Control Flow Chains

On occurrence of an external call, the allocation contexts for the corresponding assembly contexts are determined and one of these is selected based on the uniform probability distribution. The resource demands of internal

[7] As introduced in Section 11.1.2, *ExternalCallAction*, *InternalAction*, *BranchAction*, and *LoopAction* are RDSEFF-specific classes in the PCM meta-model

actions are mapped to the corresponding resources of the current resource container. Each of these resources has a scheduler. Currently, we support hard disk drive scheduled by a first-come/first-served strategy and CPU usage by processor sharing. These components are also replaceable by implementing corresponding interfaces.

Part III

Evaluation

Chapter 12

Industrial Case Study

This chapter describes a case study with an industrial enterprise application system (EAS). Kieker is integrated into the production system to monitor distributed trace and performance information in production. The SLAstic framework is used for offline analysis, namely for model extraction and performance characterization. Note that particularly Kieker has been used with additional industrial EASs (cf. Chapter 15). However, for this thesis we chose to focus on this case study as a representative EAS to be described in detail.

This chapter is structured as follows. Section 12.1 describes the conducted evaluation methodology. Section 12.2 describes the case study system. Section 12.3 details the monitoring and analysis infrastructure. The data preprocessing is described in Section 12.4. Model extraction and performance characterization are covered by Sections 12.5 and 12.6. Section 12.7 provides a summary of results. This chapter contains parts of a previous publication [van Hoorn et al., 2009c].

12.1 Evaluation Methodology

In this case study, we integrate a Kieker-based application-level instrumentation into the case study system in order to obtain control flow and performance data. This data—along with data about CPU utilization—is collected over an observation period of more than seven months and analyzed by the Kieker and SLAstic frameworks, including the techniques described in this thesis. The analyses include the extraction of architectural models as well as a performance evaluation w.r.t. varying workloads and utilization of CPUs.

The case study primarily serves to address the questions EQ1 (*Is the overall approach applicable to realistic scenarios?*) and EQ2 (*Does the approach have the desired properties?*), as introduced in Section 5.2.5.

- With respect to EQ1, we cover the following evaluation measures:
  - EM1.1 (*Confirmation of assumptions*): The obtained monitoring data is analyzed w.r.t. variations in workload intensity (EM1.1.1) and resource efficiency in terms CPU utilization (EM1.1.2).
  - EM1.2 (*Perturbation by application monitoring*): The perturbation imposed by Kieker in the production environment is evaluated on a qualitative scale.
  - EM1.3 (*Suitability of modeling language*): We evaluate whether the proposed SLAstic meta-model is able to provide an abstract view on the case study system's architecture.
  - EM1.4 (*Suitability of separating architecture and technology*): We evaluate whether the separation of architecture and technology is suitable for analyzing the case system.
- With respect to EQ2, we cover the following measures:
  - EM2.1 (*Extensibility of framework for specific purposes and technologies*): In order to use the Kieker and SLAstic frameworks for this case study, extensions for monitoring (EM2.1.2) and analysis (EM2.1.3) need to developed. By doing this, we evaluate the extensibility of our frameworks.
  - EM2.2 (*Reusability of framework*): We investigate to what degree existing parts of our frameworks can be reused for the case study, particularly in terms of modeling languages (EM2.2.1), monitoring (EM2.2.2), and analysis (EM2.2.3).
  - EM2.4 (*Suitability of MDSE techniques*): By applying the developed MDSE techniques for model extraction to the case study system, we evaluate their suitability.

Many of the measures are evaluated on a qualitative scale by demonstrating the applicability of the developed approaches based on the implemented (proof-of-concept) implementations as part of the Kieker and SLAstic frameworks.

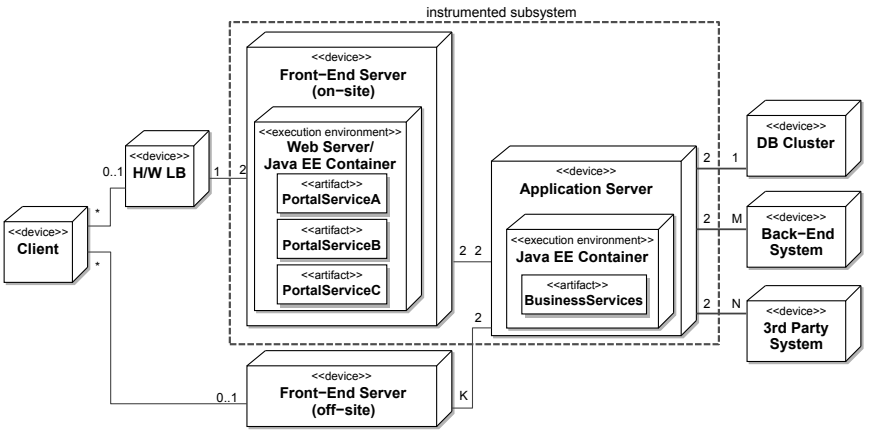


Figure 12.1. Architecture of the case study system [van Hoorn et al., 2009c].

12.2 Case Study System

The case study system is a distributed Java EE-based self-service portal that provides customers web-based access to their account data, e. g., contracts, invoices, and e-mail settings. It is maintained and operated by one of Germany's largest (almost 600,000 corporate and private customers at the end of 2011) regional providers of telecommunication services, which include Internet, telephone, and mobile.

Figure 12.1 depicts the multi-tiered system architecture. Two equally equipped (on-site) front-end servers in the presentation tier, referred to as FE0 and FE1 in this chapter, serve HTTP client requests that are distributed by an upstream hardware load balancer. Both FE0 and FE1 host three parallel portal instances, serving static and dynamic web contents. The portal instances are Java EE web applications, jointly deployed into an Apache web server/Java EE container installation. The two front-end nodes request web services from two equally-equipped application servers (AS0 and AS1) in the business-tier. The web service requests from FE0 and FE1 to AS0 and AS1 are load-balanced in a round-robin fashion, employing the Apache web

server's *mod_proxy_balancer* module.[1] The Java EE application logic on all four nodes is implemented using the Spring framework and the Apache CXF web service technology. The business-tier nodes access a replicated cluster of database servers as well as various services from the business tier, e. g., via web service or EJB calls. For a brief introduction into common architectures and technologies used by Java EE-based EASs, please refer to Section 3.3.1.

12.3 Monitoring and Analysis Infrastructure

The two front-end nodes (`FE0` and `FE1`) and the two application server nodes (`AS0` and `AS1`) are equipped with monitoring facilities to acquire and collect *a*) the CPU utilization for each of the four server nodes and *b*) distributed application-level trace information, as described in Section 7.2. The monitoring infrastructure for the CPU utilization was already in place when starting the case study. We developed an importer that enables the integration of these CPU measurements into Kieker and SLAstic. In order to monitor distributed trace information, we refined and integrated Kieker. Sections 12.3.1 and 12.3.2 detail the monitoring facilities. As detailed in Section 12.3.3, the monitored data is processed offline, employing the Kieker and SLAstic frameworks.

12.3.1 Monitoring of CPU Utilization

On the two front-end nodes (`FE0` and `FE1`) and the two application server nodes (`AS0` and `AS1`), the CPU utilization is continuously measured and logged using RRDtool. In order to process the data with Kieker and SLAstic, we developed a Kieker-based importer program.

RRDtool Configuration

RRDtool (short for Round Robin Database tool) is a popular open-source tool for storing (and plotting) aggregated time series data in so-called Round Robin Databases (RRDs). An RRD is configured to accept input data

[1] `http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html`

for one or more named *data sources* in periodic time steps for a given *base interval*. The values for a time step, called *primary data points*, are passed to Round Robin Archives (RRAs) that are contained in an RRD and aggregate a configured number of primary data points into a *consolidated data point* using one of the available consolidation functions (average, maximum, etc.). These consolidated data points are archived in the RRA's circular storage allocated for each of the data sources. The RRDtool web site includes a tutorial introducing basic concepts and examples of use.[2]

Table 12.1 lists the configuration of the RRD used on each of the nodes in the case study. Four data sources exist for fractions of time a CPU spends in *a*) user space, *b*) system (i. e., kernel) space, *c*) idle mode, and *d*) waiting for I/O operations. An external script provides corresponding CPU utilization measurements with a base interval of 300 seconds. This data is stored as time series by four RRAs. RRA 0 keeps the individual measurements of the past 400 days. Note that the consolidation function (maximum) has no effect, because it is applied to a single value. RRAs 1–3 store average values with a resolutions of 0.5, 2, and 24 hours for a configured number of days (Table 12.1b).

Data Import into Kieker

We developed an importer program that converts the CPU utilization data from RRA 0 (for each server node) into a Kieker (file system) monitoring log of `CPUUtilizationRecords` (Figure 7.2), enabling further processing with the Kieker and SLAstic frameworks. In an intermediate step, a Bash script creates one CSV file per server node—each row including a 300 second step with the corresponding values from RRA 0 for each of the data sources, i. e., *user*, *system*, *idle*, and *io*. A Java program employs Kieker to import the CSV data into a monitoring log of `CPUUtilizationRecords`. One `CPUUtilizationRecord` is created for each CSV row. Table 12.2 lists which values are assigned to individual `CPUUtilizationRecords` based on corresponding data from the server node's RRA 0. The fields *idle*, *user*, and *system* are computed from the values of the corresponding data sources. The *totalUtilization* is computed by subtracting the *idle* value from 1. Because the data source *io* is available only on the front-end nodes and the values of the data sources closely but

[2] `http://oss.oetiker.ch/rrdtool/`

Table 12.1. Configuration for each of the RRDs and the contained RRAs

(a) RRD base interval and data sources (of equal type and heartbeat value)

Base interval	300 sec. (i. e., 5 min.)
Data sources	“user”, “system”, “idle”, “io” | *type:* Gauge | *heartbeat:* 600 sec.

(b) Configuration of the RRAs contained in the RRD. The two bottom lines list properties derived from the upper parameter values together with the information that a primary data points maps to a 5 minutes base interval (Table 12.1a). The consolidation function of RRA 0 has no effect, as it is applied to single values.

RRA parameter	**RRA 0**	**RRA 1**	**RRA 2**	**RRA 3**
Consolidation function	(Maximum)	Average	Average	Average
Rows	115200	700	775	797
Primary data points per row	1	6	24	288
Time interval per row	5 min.	30 min.	120 min.	24 hours
Capacity (days)	400	14.58	64.58	797

Table 12.2. Assignment of values to `CPUUtilizationRecord` fields (*hostname* omitted). Variables indexed by $_{RRA_0}$ refer to the values of the corresponding data source values contained in RRA 0.

Field	**Value**	**Field**	**Value**
timestamp	timestamp$_{RRA_0}$	*system*	system$_{RRA_0}$
cpuID	"cpu0"	*wait*	totalUtilization - user - system
idle	idle$_{RRA_0}$	*nice*	0.0
totalUtilization	1.0 - idle	*irq*	0.0
user	user$_{RRA_0}$		

not exactly sum up to the value 1, we decided to approximate the *wait* value, which is the corresponding field in the records, by subtracting *user* and *system* from the total utilization value. The *hostname* is set to the name of the respective host.

12.3.2 Monitoring of Trace Information

Six *Kieker.Monitoring* instances on the front-end nodes (one for each portal) in addition to one *Kieker.Monitoring* instance per business-tier node, result in a total number of eight concurrent *Kieker.Monitoring* instances distributed over the four server nodes. Each *Kieker.Monitoring* instance is configured to use an asynchronous file system writer, as described in Section 7.3, which writes the *Monitoring Log* to the local file system.

Six different *Monitoring Probe* types, detailed below, are integrated on three layers of the software architecture—namely, Servlet-based HTTP request entry point, Spring-based business logic, and CXF-based web services—to jointly monitor distributed *execution traces* across the server nodes of the front-end and business-tier. *OperationExecutionRecords* are used as the *Monitoring Record* type representing the data of an *execution*, as described in Sections 7.2 and 7.3.

- *Servlet probes.* A *Monitoring Probe* on the front-end nodes intercepts incoming HTTP service requests and initializes the trace and session information for this request, including a unique trace identifier, as well as the initialization of the *execution order index* and *execution stack size* values (see Section 7.2). After the execution of the actual request, the *Monitoring Probe* resets the trace information and logs the *execution* on the HTTP request level. This *Monitoring Probe* had been developed in a previous case study during the course of this thesis and has been refined as part of this case study.
- *Spring probes.* A Spring-based *Monitoring Probe* intercepts and logs executions of the business service implementations on the business-tier nodes. This *Monitoring Probe* had been developed in a previous case study during the course of this thesis and has been refined as part of this case study.
- *CXF probes.* Two CXF *Monitoring Probes* on the front-end nodes are used to intercept outgoing web service calls as well as the corresponding responses to and from the business-tier. For each outgoing web service call, the trace and session identifiers, as well as the *execution order index* value are integrated into the corresponding SOAP message. A *trace*'s *execution order index* value is updated according to the value contained in the response message sent by the business-tier node. Two corresponding CXF *Monitoring Probes* are integrated into the business-tier nodes to

Table 12.3. Basic statistics about the (raw) Kieker monitoring logs with operation executions

Node	Size	Observation Period	# Logs	# Files	# Executions
FE0	3.8 GB	Nov. 26, 2009 – July 11, 2010	51	870	18,358,970
FE1	3.9 GB	Nov. 26, 2009 – July 11, 2010	42	866	18,482,244
AS0	3.2 GB	Dec. 06, 2009 – July 11, 2010	47	799	17,183,364
AS1	3.3 GB	Dec. 06, 2009 – July 11, 2010	31	798	17,280,534

manage the trace and session information of incoming and outgoing SOAP messages. A web service call results in a logged *execution* on the calling front-end node and a logged *execution* on the called business-tier node. These *Monitoring Probes* have been developed as part of this case study.

12.3.3 Offline Analysis

In this case study, the Kieker and SLAstic frameworks are employed in the framework deployments *1. (online/production)* and *4. (offline/lab)* described in Section 8.6.2 and depicted in Figures 8.6a and 8.6d. The production monitoring data is written to a file system *Monitoring Log*, which is processed offline by Kieker and SLAstic. Kieker's analysis features are used for basic trace analysis and visualization. SLAstic serves to extract a SLAstic model instance of the case study system, as well as to process the workload and performance measurements. For the latter, we additionally use the statistical language and environment *R* [R Development Core Team, 2014] with respective scripts processing the data provided by SLAstic.

12.4 Data Preprocessing

Table 12.3 lists basic statistics from the monitoring logs collected from the four server nodes. A first version of Kieker was deployed on November 26,

2009 to the front-end nodes. The instrumentation of the business-tier nodes was deployed on December 6, 2009. During the course of the subsequent weeks, we deployed refined versions of the Kieker-based instrumentation, particularly regarding the distributed tracing functionality via web services. The observation period for all nodes ends on July 11, 2010. Table 12.3 includes for each server node the number of Kieker monitoring logs, the number of contained Kieker log files, and the number of monitored operation executions—obtained by counting the lines in the log files. Note that each of the two front-end nodes includes three parallel portal instances, each having its own monitoring logs. In the further analysis, we do not distinguish between the three portal instances. A new monitoring log is created each time the respective application or portal instance is (re)started. A first attempt to process the raw data with Kieker's trace analysis revealed that three corrupted monitoring records are included in the monitoring logs of the business-tier nodes. These records were removed. CPU utilization measurements obtained by the RRDtool (RRA 0) are available for the 400-day period (cf. Table 12.1a) from June 14, 2009 to July 19, 2010.

Kieker's trace analysis tool detected 23,804,565 traces from which 23,551,575 (98,94%) could be reconstructed successfully. The trace detection timeout was configured to five hours based on the memory consumption needed for the trace reconstruction. (A higher timeout value leads to increased memory usage because executions within a trace are stored for a longer period of time, waiting for additional executions of the same trace.) We performed a basic investigation of the distribution of broken traces over time. The results are depicted in Figure 12.2. It can be observed that the number of invalid traces is particularly high during the time that only the front-end nodes are instrumented. A second drop in the number of broken traces can be observed at the beginning of March.

For the further analysis, we decided to create a refined monitoring log in which all executions belonging to broken traces are removed. An execution of Kieker's trace analysis tool on the refined monitoring log resulted in 23,551,569 successfully reconstructed traces (100%). The numbers of valid traces over time are included in Figure 12.3a. Also included in the figure is the average maximum execution stack size per trace. In addition to this, Figure 12.3b lists the frequencies of occurrence for the observed maximum execution stack sizes (ess).

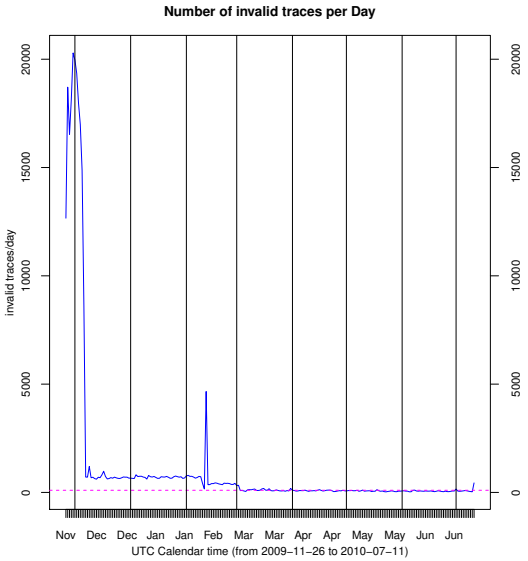


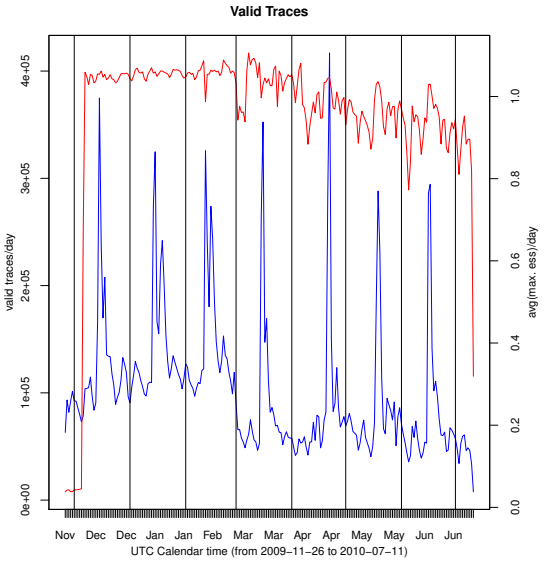
Figure 12.2. Number of invalid traces (per day) during the observation period

12.5 Model Extraction

Sections 12.5.1 and 12.5.2 describe the architectural models of the case study system extracted by the Kieker and SLAstic frameworks.

12.5.1 Kieker Model

We used Kieker's standard trace reconstruction and analysis features to extract and visualize architectural models conforming to the system and trace meta-model described in Section 7.2.1 (cf. [Kieker Project, 2014a]). The reconstructed architecture comprises 10 component types, 110 operations, 10 assembly components, 4 execution containers, and 20 deployment components. The analysis took 30 minutes on a Sun Blade X6270 (cf. Figure 13.1). Figure 12.4 shows generated calling dependency graphs of the architectural models on different abstraction levels.



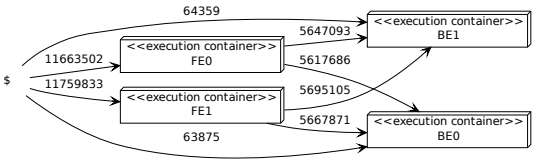
(a) Valid traces and ess values over time

Max. ess	0	1	2	3	4
# traces	9,343,275	8,269,224	2,848,574	3,090,487	9

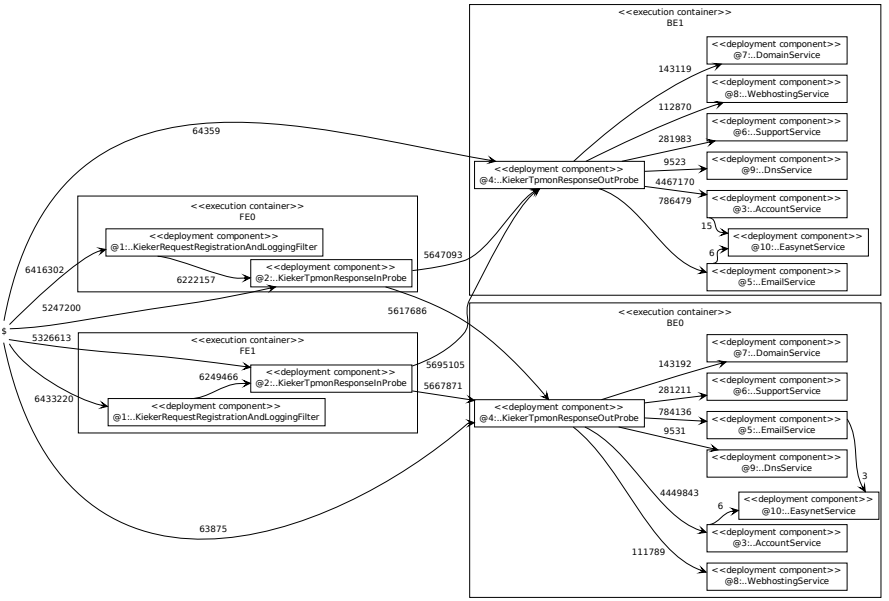
(b) Frequencies of maximum ess value

Figure 12.3. Number of valid traces (per day) and average maximum execution stack sizes (red in (a)) during the observation period

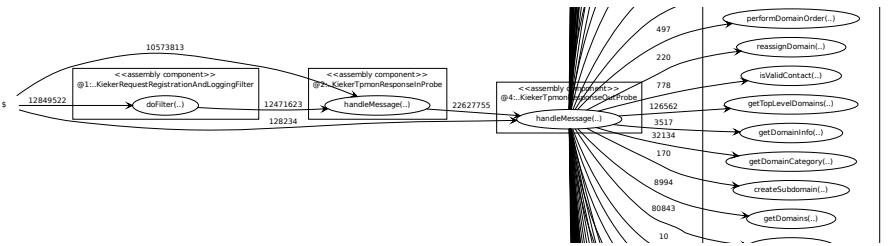
- Figure 12.4a depicts calls to and between the four nodes. Corresponding to the system architecture in Section 12.2—and depicted in Figure 12.1 (dashed part)—the two-layer architecture with two front-end and two business-tier nodes can be observed. Also visible are the load-balanced accesses to/between the front-end and business-tier nodes. As described in Section 12.2 the business-tier nodes are not only accessed via the on-site front-end nodes but also by other (off-site) systems (cf. Figure 12.1). In total, the number of external calls sums up to 23,551,569, which corresponds to the aforementioned number of extracted traces reported by Kieker's trace analysis.



(a) Execution container dependency graph



(b) Deployment-level component dependency graph (slightly re-arranged)



(c) Assembly-level operation dependency graph (excerpt)

Figure 12.4. Visualizations of selected architectural models reconstructed by Kieker

- Figure 12.4b shows the deployment-level component dependency graph of the system, comprising the 20 deployment components (10 assembly components/component types) and their weighted calling dependencies. Note that three of the component types (including `Kieker` in the name) were introduced by Kieker's monitoring probes, namely the Servlet filter and the CXF probes for outgoing and incoming web service calls. In addition to the expected requests to the front-end nodes via the Servlet probe (`KiekerRequestRegistrationAndLoggingFilter`), direct calls to the probe that monitors outgoing web service calls (`KiekerTpmonResponseInProbe`) can be observed. These calls are caused by asynchronous business-tier requests by the web portals in the front-end nodes.

- Figure 12.4c shows an excerpt of the assembly-level operation dependency graph. It can be seen that the three components introduced by the Kieker instrumentation each have a single operation. In each case, the weights of the calls are the sum of the calls per deployment component. The depicted operations in the business tier belong to the component `DomainService` (cf. Figure 12.4b).

12.5.2 SLAstic Model

We applied our approach for extracting a SLAstic model from the Kieker monitoring log, employing the MDSE-based techniques detailed in Section 9.3. Details on the SLAstic system and usage model are described in the remainder of this section.

The analysis took 310 minutes on the same machine that has already been used for the Kieker analysis. Note that the long duration is a consequence of various SLAstic features (Section 8.6.4) being enabled, e. g., the rewriting of the entire Kieker monitoring log by SLAstic's Monitoring Manager, and the extraction of time series for various performance measures. A current (performance) limitation in the implementation of the latter feature requires the time series files to be closed after each entry due to the large number of open file handles (794 in this case). Without the aforementioned features and the usage model extraction being disabled, the analysis took about 60 minutes.

System Model

For the type and operation signature name abstraction (Section 9.3.3), the mode *class strict* (page 158) is used, i. e., component types refer to Java classes. As a consequence, the architectural structure is close to the structure included in the reconstructed Kieker model (described in Section 12.5 and visualized in Figure 12.4). Corresponding to the reconstructed architectural entities in the Kieker model, the reconstructed SLAstic model includes 10 component types, 110 operations, 10 assembly components, 20 deployment components, and 4 execution containers. However, as the Kieker meta-model includes only a subset of the concepts from the SLAstic meta-model, additional entities are included, e. g., 10 interfaces, 110 signatures, 4 execution container types, 3 system delegation connectors, and 10 assembly component connectors. Each execution container type includes a resource specification for a single CPU; the resource instance is included in each of the respective four execution containers.

Due to the naming rules of the extraction procedure (Section 9.3.2), the entity names slightly deviate from those in the Kieker model. This mainly concerns prefixes and suffixes, e. g., `SupportService_T`, `ISupportService_T` (), and `FE0_T` are names for a component type, an interface, and an execution container type in the SLAstic model. Due to the architectural structure being similar to the Kieker model, we omit a visual representation of the SLAstic model.

Usage Model

As described in Section 6.3.3, the usage model currently includes three types of usage information, namely *a*) calling relationships between operations of component types and signatures of interfaces, *b*) operation call frequencies, as well as *c*) assembly connector call frequencies.

- An interesting property w.r.t. the (111) calling relationships can be observed for the calling frequency distribution between the Servlet filter's `doFilter` operation and the `handleMessage` operation for outgoing web service calls. A histogram of the frequency distribution is shown in Figure 12.5. A single call of the `handleMessage` operation may lead to up to 932 web service calls to the business-tier nodes. Note that the `doFilter`

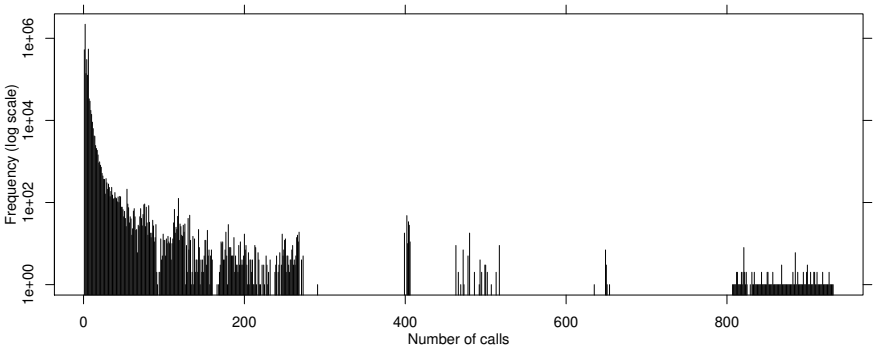


Figure 12.5. Distribution of calling frequencies from the *doFilter* operation to the *handleMessage* operation

operation does not call `handleMessage` directly: the `doFilter` is an interceptor for incoming service requests; requests are handled by parts of the portal that are not instrumented and issue the web service requests. All other calling relationships have a deterministic number of calls: 1 in all other cases except for one, which is 3. Note that the information about zero calls is not explicitly included in the usage model but needs to be computed by also considering the total number of executions of the callee operation.

- The frequencies of execution for the 110 operations vary between 1 and 23,045,436 (mean 638,471; quartiles 275/2,582/21,807). The three most frequently executed operations are the `handleMessage` operations of outgoing (23,045,436 executions) and incoming (22,755,989) web service calls, as well as the Servlet filter's `doFilter` operation (12,849,522). Execution frequencies of selected operations are included in Figure 12.4c. The distribution of execution frequencies across the different component types is included in Figure 12.4b.
- The call frequencies for the system-provided delegation connectors (corresponds to the entry-level system calls) can be obtained from Figure 12.4b, namely 12,849,522, 10,573,813, and 128,234 for the three different entry points (Servlet filter and web service interceptors).

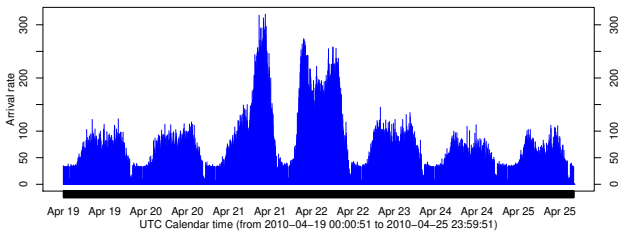


Figure 12.6. Arrival rates of the Servlet entry (assembly-level) over one week

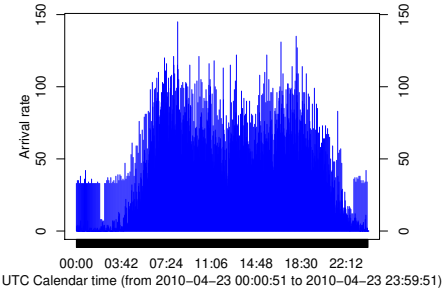
12.6 Performance Characterization

The following two sections describe the characterization of the workload (Section 12.6.1) and the CPU utilization (Section 12.6.2).

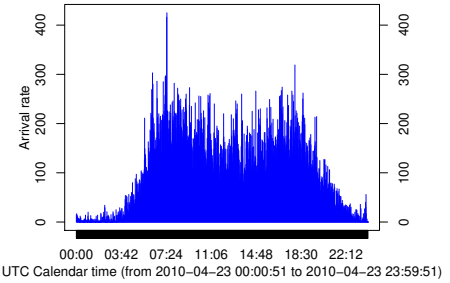
12.6.1 Workload Characterization

As introduced in Section 4.1.2, workload characteristics are often divided into measures concerning *workload intensity* and *resource demands*. Some information on the workload intensity has already been reported in Figure 12.3a—in that case the number of valid traces that were executed by the system during each day of the observation period. In this section, the focus of workload characterization will be on workload intensity in terms of arrival rates—to be understood as the number of requests to a software service observed over a specified time period (cf. page 41). The SLAstic framework already reports arrival rates for different architectural entities (software services) in the SLAstic model, namely for assembly components, for deployment components, and for operations. In order to study the characteristics of how logical components of the case study system are used, we will consider arrival rates for assembly components. Remind that Figure 12.4b depicts the *deployment* components of the case study system. In this case, always two deployment components of the same type are grouped into a single assembly component (10 in total, cf. Section 12.5.2). The arrival rate for an assembly component is the sum of arrivals to any of the operations of any of the (two) deployment components.

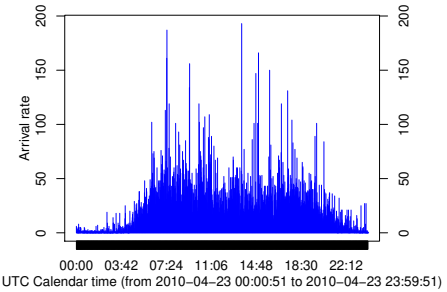
Selected results are shown in Figures 12.6 and 12.7. For this analysis,



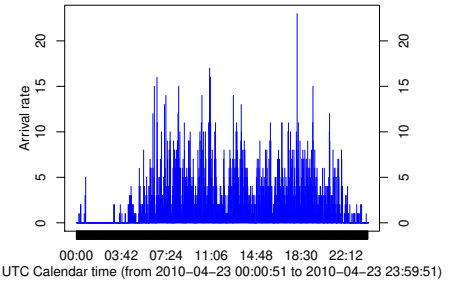
(a) Servlet entry (one day)



(b) Business-tier entry (one day)



(c) *AccountService* (one day)



(d) *EmailService* (one day)

Servlet	0.766 ± 0.022	0.602 ± 0.034	0.507 ± 0.039
	Business tier	0.810 ± 0.019	0.507 ± 0.039
		AccountService	0.422 ± 0.043
			EmailService

(e) Correlation coefficients (Pearson, 95% CI) for pairs of arrival rates in (a)–(d)

Figure 12.7. Arrival rates and correlations for selected assembly components

the SLAstic framework was configured to compute the arrival rate in 60-second intervals. Note that the times are given in UTC, while the case study system—and probably most of its users—is located in CET (UTC+1) and CEST (UTC+2) respectively. Figure 12.6 depicts the arrival rates of requests to the front-end nodes that have been observed during one week (Monday to Sunday), selected from the overall observation period. Particularly, these are the requests also processed by Kieker's Servlet filter (Figure 12.4). A typical seasonal usage pattern for EASs can clearly be observed also for this system: each day, the arrival rate increases during the morning hours and decreases in the evening, leading to very low arrival rates during the night; during the lunchtime period, the arrival rates decrease. During this week, the arrival rates on Wednesday and Thursday are considerably higher than on the other days of the week.

Figures 12.7a to 12.7d show the arrival rates for a selected day (Friday) from this week, namely the Servlet entry point (Figure 12.7a, showing an excerpt from Figure 12.6), the business-tier entry point (Figure 12.7b), as well as the business-tier services `AccountService` (Figure 12.7c) and `EmailService` (Figure 12.7d). Note that these services, in addition to the web service component in the front-end, are the four most frequently called services. The aforementioned usage pattern over a day can be observed in the plots for this specific day.

We were interested to what extent the requests to services in the case study system show a linear relationship, i. e., whether the arrival rates are proportional to each other or whether they are (also) impacted by other factors. As a measure of correlation, we computed the Pearson correlation coefficients for each pair of arrival rates from Figures 12.7a to 12.7d. The resulting coefficients along with the 95% confidence intervals are listed in Figure 12.7e. The values are roughly in a range between 0.4 and 0.8, indicating that a certain level of linear relationship exists in each case. This is supported by the visual impressions that can be drawn from Figures 12.7a to 12.7d. High coefficients can be observed between the business-tier entry and the `AccountService` (0.810) as well as between the Servlet entry and the business-tier entry (0.766). Medium values can be observed between Servlet entry and `AccountService` (0.602), as well as between Servlet/business-tier and `EmailService` (both 0.507). The lowest values can be observed between `AccountService` and `EmailService` (0.422). A high correlation value between Servlet entry and business-tier entry was expected as Servlet entry calls

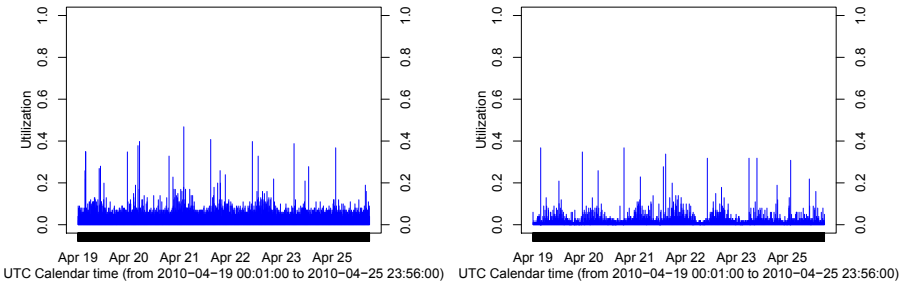
also lead to business-tier requests. However, as described in Section 12.5.2 (usage model), the number of calls is not deterministic and business-tier calls may also result from asynchronous front-end requests as well as direct business-tier requests (cf. Figures 12.1 and 12.4b). Both the *AccountService* and the *EmailService* are only called via the business-tier entry point, i. e., a direct dependency exists in these cases. Based on the data from the usage model, it can be computed that the ratio of the number of *AccountService* and business-tier calls is 40%; for the *EmailService*, the ratio is 7%. The high ratio for business-tier/*AccountService* may explain the high correlation value for this pair of services. The indirect call paths from the Servlet entry—via the business-tier entry—can explain the medium correlation with the *AccountService*. The *EmailService* shows a lower linear relationship to the other services—particularly, to the *AccountService* located at the same architectural layer.

12.6.2 Characterization of CPU Utilization

This section summarizes the results of analyzing the CPU utilization of the four server nodes. As described in Section 12.3.1, different CPU utilization measures (cf. Table 12.2), including the total CPU utilization (*totalUtilization*) and its components (*user*, *system*, etc.), were imported from RRDtool into Kieker and the SLAstic framework. The resolution of measurements is five minutes. In this section, we focus on the total CPU utilization measurements.

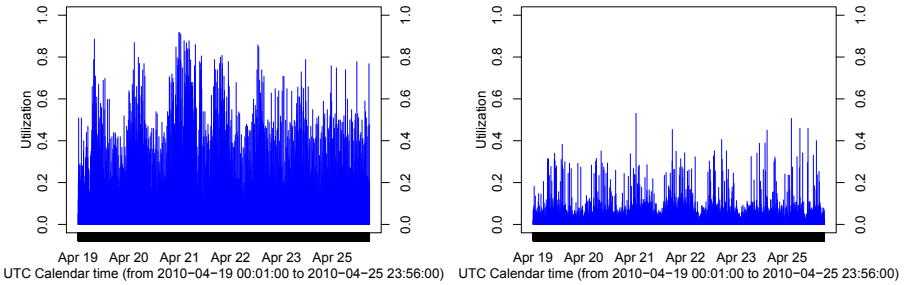
Figure 12.8 shows the CPU utilization of the four servers over one week (Figures 12.8a to 12.8d) and over one day in this week (Figures 12.8e to 12.8h). The time periods (week/day) correspond to the periods considered for the workload characterization in Section 12.6.1. In addition to this, Figure 12.9 shows statistics and probability density plots of the observed CPU utilization measurements.

A seasonal pattern can be observed for the CPU utilization over the week in that the seven days can be identified (higher utilization during the day; lower utilization during the night). However, this pattern is not as distinct as for the arrival rates (e. g., Figure 12.7). Note that due to the high resolution of measurements (five minutes), the higher utilization values dominate the lower utilization values in the plots. Figures 12.9a and 12.9b depict the probability density functions for the utilizations. As listed in



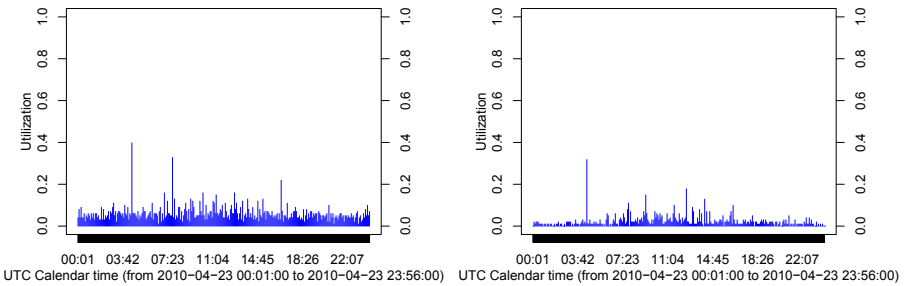
(a) FE0 (one week)

(b) FE1 (one week)



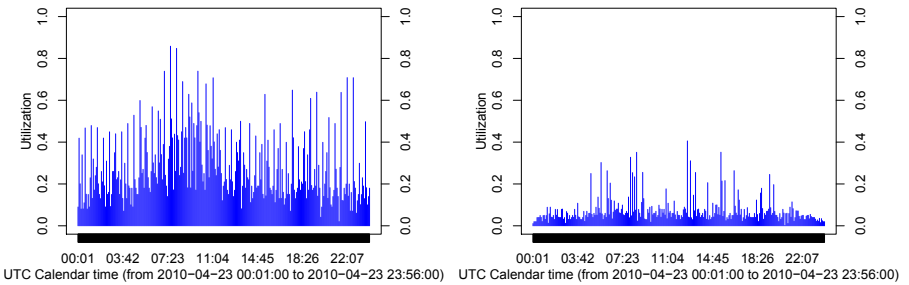
(c) AS0 (one week)

(d) AS1 (one week)



(e) FE0 (one day)

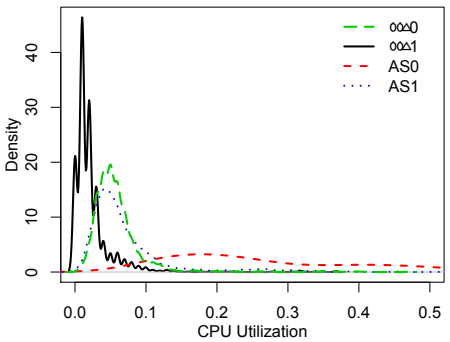
(f) FE1 (one day)



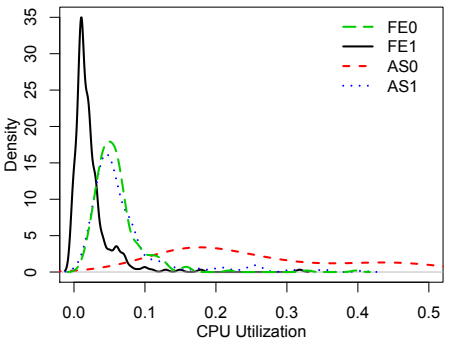
(g) AS0 (one day)

(h) AS1 (one day)

Figure 12.8. CPU utilization (5 minute intervals) of the four servers over one week ((a)–(d)) and over a selected day ((e)–(h))



(a) Density for Figures 12.8a to 12.8d (week)



(b) Density for Figures 12.8e to 12.8h (day)

	Quantiles							
	min	**0.25**	**0.50**	**0.75**	**0.95**	**max**	**mean**	**sd**
FE0	0.00	0.04	0.05	0.07	0.11	0.47	0.06	0.04
FE1	0.00	0.01	0.02	0.03	0.07	0.37	0.02	0.03
AS0	0.01	0.16	0.24	0.40	0.65	0.92	0.29	0.17
AS1	0.00	0.04	0.05	0.08	0.22	0.53	0.07	0.06
mean	0.02	0.07	0.10	0.14	0.34	0.21	0.11	0.05

(c) Summarizing statistics (week). The last row shows the statistics for the mean CPU utilization of all servers per observation time.

Figure 12.9. Statistics and probability density functions for CPU utilization

Figure 12.9c, front-end server FE1 has the lowest mean CPU utilization (2%), followed by front-end server FE0 (6%) and application server AS1 (7%). The measurements for FE0 and AS0 have a similar distribution. The CPU utilization of application server AS0 shows a considerably higher mean value (29%) and standard deviation. Relative to their mean values, the standard deviations for the other three servers is also quite high.

In general, it can be concluded, that the CPU utilization of all four servers—also in light of the higher measurements for AS0—is extremely low. This is particularly true for both front-end nodes FE1 and FE0, whose utilization is less or equal 0.03 and 0.07 respectively for 75% of the time, as well as less or equal 0.07 and 0.11 respectively for 95% of the time. Looking at the mean CPU utilization of all four servers every 5 minutes, it can be

observed that this value is less or equal 0.14 during 75% of the time and less or equal 0.34 during 95% of the time (last row in Figure 12.9c).

12.7 Summary of Results

In this case study, we employed the Kieker and SLAstic frameworks to instrument, monitor, and analyze a typical distributed Java EE-based enterprise application system (EAS). The Kieker-based instrumentation was used to monitor performance and trace information. Monitoring data from production use was collected over a period of more than seven months. This data was (pre-)processed by the Kieker and SLAstic frameworks for performance and workload characterization, as well as extraction of Kieker and SLAstic models. We observed and quantified varying workloads and low CPU resource utilization in the production data.

With respect to the addressed evaluation questions EQ1 (*Is the overall approach applicable to realistic scenarios?*) and EQ2 (*Does the approach have the desired properties?*) we come to the following conclusions:

- EQ1: The performance evaluation revealed that the system is exposed to a varying workload intensity (EM1.1.1) and that the system's resources in terms of CPU utilization is very low (EM1.1.2). No perceivable performance overhead and stability issues caused by our monitoring were reported in this case study (EM1.2). We experienced that both the extracted Kieker and SLAstic models provide useful views to the case system's architecture in terms of structure and behavior (EM1.3). With respect to the structure, the web service functionality is automatically extracted as architectural components. However, based on a model refinement, they could be transformed into connector elements. Note that when using our model-driven instrumentation approach described in Section 9.1, this manual refinement would not be necessary. The separation of architecture and technology turned out to work well for this case study (EM1.4). Note that we had no access to system source code and deployment environment but were able to extract the architectural information solely based on the monitoring data.
- EQ2: The frameworks' extensibility (EM2.1) and reusability (EM2.2) could be exploited to a high degree. Several extensions to Kieker were

developed as part of this case study, e. g., *Monitoring Probes* for distributed tracing via web service calls. The RRD-based monitoring infrastructure for CPU utilization, which was already present in the system, was integrated into Kieker by developing a respective adapter that transforms the data into respective Kieker *Monitoring Record*s. Large parts of the existing framework parts could simply be used or have been further refined for reuse—also for/from further case studies and lab experiments. The MDSE techniques could successfully be employed to extract architectural models of the case study system (EM2.4). The offline analysis of the data set required a considerable amount of processing time. This has also been investigated by Fittkau et al. [2013] who developed first improvements. However, for our offline analysis in the case study setting, Kieker's trace analysis performance was sufficient.

Threats to Validity Threats to external validity particularly concern the size of the of the case study system and the fact that we considered only a single system in this chapter. We are aware that EASs of much larger size exist, consisting of hundreds or thousands of servers instead of the four servers in this systems. However, we argue that the considered system can be considered a representative for this class of EASs. We conducted similar case studies with other production systems—particularly in the Kieker context, as detailed in Chapter 15. Also, Kieker has been integrated into a company's MDSE platform conforming to the approach by Stahl and Völter [2006]. With respect to internal validity, we assessed most of the qualitative measures by evaluating our own approach, e. g., its extensibility and reusability. However, we argue that similar reuse and extension of our approaches has been conducted by other researchers and practitioners, as, for instance, detailed in Chapter 15. Other internal threats include that *a*) different Kieker versions were used throughout the observation period, *b*) broken traces traces were removed from the monitoring log, and *c*) only a single week of the observation data was used for the performance characterization. We did not fully analyze the reason for the occurring broken traces. However, as distinct steps of decreasing broken traces can be observed in the data (Figure 12.2), we assume that refined Kieker versions fixed the problem. As mentioned in Section 12.4, the number of broken traces compared to the total number of traces ($\approx 1\%$) is negligible—

particularly, knowing that the majority of these traces occurred in the beginning of the observation period with a second decrease in March. The selected week for the workload characterization is from April, i. e., contains a low number of broken traces. In our opinion, the selection of a single week does not have a big impact on the results, as a varying workload intensity can be observed in Figure 12.3a (number of valid traces per day) for the complete observation period. The selected week in April includes an average workload intensity compared to the rest of the observation period (looking at the values starting from March).

Chapter 13

Lab Experiments

This chapter presents the results of applying our developed approach in a lab experiment. SLAstic is employed to control the capacity of a Java EE-based sample application deployed to an Eucalyptus-based IaaS cloud environment (Section 3.3.2).

This chapter is structured as follows. Section 13.1 describes the conducted evaluation methodology. Sections 13.2 to 13.4 present the experimental setting, the developed framework extensions, and the experimental results. Section 13.5 provides a summary of results for the lab experiments. This chapter includes contents of joint work with Fittkau [2011] and Huber et al. [2012, 2014] (cf. Section 5.3).

13.1 Evaluation Methodology

In our experiment, we expose a Java-based enterprise application, deployed to a Eucalyptus-based IaaS cloud infrastructure, to a synthetic probabilistic workload with varying intensity based on a 24-hour workload profile obtained from an industrial system. The SLAstic framework is used to control the capacity of the application, aiming for increased resource efficiency. Based on a rule-based adaptation policy, application instances are added and removed based on continuously obtained performance measurements about the workload intensity. The experiment is executed in two scenarios—with (Scenario 1) and without (Scenario 2) adaptive capacity management.

This experiment primarily serves to address the questions EQ1 (*Is the overall approach applicable to realistic scenarios?*), and EQ2 (*Does the approach have the desired properties?*), as introduced in Section 5.2.5.

- With respect to EQ1, we cover the following evaluation measures:
  - EM1.2 (*Perturbation by application monitoring*): The monitoring overhead imposed by Kieker is evaluated on a qualitative scale.
  - EM1.3 (*Suitability of modeling language*): We evaluate whether the proposed SLAstic meta-model is able to provide an abstract view on the sample system's architecture.
  - EM1.4 (*Suitability of separating architecture and technology*): We evaluate whether the separation of architecture and technology is suitable for the IaaS-based lab environment.
- With respect to EQ2, we cover the following measures:
  - EM2.1 (*Extensibility of framework for specific purposes and technologies*): In order to use the Kieker and SLAstic frameworks for this experiment, extensions for monitoring (EM2.1.2), analysis (EM2.1.3), and reconfiguration (EM2.1.4) need to developed. By doing this, we evaluate the extensibility of our frameworks.
  - EM2.2 (*Reusability of framework*): We investigate to what degree existing parts of our frameworks can be reused for the lab experiment, particularly in terms of modeling languages (EM2.2.1), monitoring (EM2.2.2), analysis (EM2.2.3), and reconfiguration (EM2.2.4).
  - EM2.3 (*Suitability of reconfiguration operations*): We integrate the SLAstic framework with the Eucalyptus infrastructure, including a technology-specific implementation of the SLAstic reconfiguration operations deallocation and allocation of execution containers, as well as replication and dereplication of software components (Chapter 10). A quantitative evaluation investigates their potential to increase resource efficiency (EM2.3.1) and their transparency (EM2.3.2).
  - EM2.4 (*Suitability of MDSE techniques*): By applying the developed MDSE techniques for model extraction to the lab system, we evaluate their suitability.

As for the case study (Chapter 12), many of the measures are evaluated on a qualitative scale by demonstrating the applicability of the developed approaches based on the implemented (proof-of-concept) implementations as part of the Kieker and SLAstic frameworks.

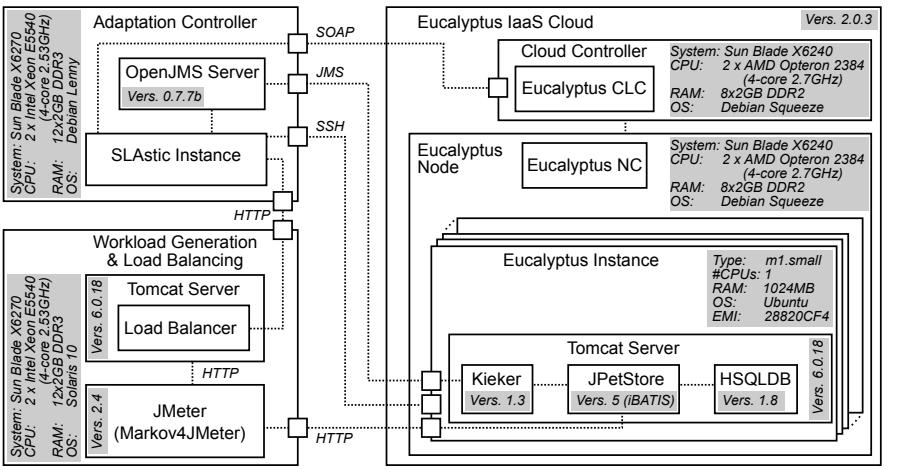


Figure 13.1. Overview of the experiment infrastructure

13.2 Experimental Setting

Section 13.2.1 provides a detailed description about the software and hardware infrastructure, including the instrumented sample enterprise application, its deployment to the Eucalyptus-based IaaS infrastructure (see page 33), as well as the workload generator and the integration of the SLAstic framework for online capacity management. The two experiment scenarios, including the workload curve and the rule-based adaptation strategy, are detailed in Section 13.2.2. Note that the extensions for the SLAstic framework that have been developed for this experiment are described in Section 13.3.

13.2.1 Software and Hardware Environment

Figure 13.1 depicts the technical experiment infrastructure in terms of hardware and software components as well as connections among these—including additional information on hardware equipment, version numbers, and protocols.

In total, the infrastructure comprises four physical machines, which are part of the Software Performance Engineering Lab (SPEL) at Kiel University's Software Engineering Group.[1] A first machine (Adaptation Controller) hosts an instance of the SLAstic framework that controls the adaptation and receives Kieker monitoring records via the JMS server deployed to the same machine. A second machine (Workload Generation and Load Balancing) hosts the workload generator and the load balancer. The Eucalyptus cloud is hosted by the remaining two machines—one of which (Cloud Controller) hosts the Eucalyptus Cloud Controller (CLC) that manages Eucalyptus instances being dynamically created and released on the other machine (Eucalyptus Node). Each Eucalyptus instance hosts a Kieker-instrumented instance of the JPetStore sample application, connected to an HSQLDB DBMS. The maximum number of parallel Eucalyptus instances is limited to eight.

The remainder of this section provides further details on *a*) the JPetStore sample application, *b*) the load balancing approach, *c*) and the workload generation.

JPetStore Sample Application The iBATIS JPetStore 5 is a distributed Java EE application that represents a typical online shopping system—in this case offering pets. An HTML-based web interface enables to perform typical use cases, such as signing in and off, browsing through the product catalog, maintaining a virtual shopping cart, and purchasing an order. The technical architecture comprises a common 3-tier structure for web applications with a presentation layer, a service layer, and persistence layer connected to a DBMS. Building on Java EE web technologies, the JPetStore needs to be deployed to an Application Server (Section 3.3.1)—in our case an Apache Tomcat Servlet container. The persistence layer connects to a relational DBMS—in our case HSQLDB. A more detailed description of the application w.r.t. its use cases and technical architecture can be found in our previous works [van Hoorn, 2007; van Hoorn et al., 2008].

The iBATIS JPetStore has been developed to demonstrate the capabilities of the Apache iBATIS persistence framework. Since its first use as part of my Diploma thesis [van Hoorn, 2007], we have used the application for experiments in many contexts—including teaching and research (e. g., [van

[1] http://www.se.informatik.uni-kiel.de/en/research/software-performance-engineering-lab-spel/

Hoorn et al., 2008; Gul et al., 2008; Marwede et al., 2009; Rohr et al., 2010]). In the meantime, iBATIS has been retired and superseded by a successor, called MyBatis. The respective successor of iBATIS JPetStore 5 is the MyBatis JPetStore 6. However, despite of the replaced persistence framework, the application has experienced no major changes. For quite some time already, the JPetStore (originally the iBATIS version; in the meantime, the MyBatis version) is part of each Kieker release to demonstrate its use for Java EE-based applications in the user guide [Kieker Project, 2014a]. The version used in this experiment is the version included in the Kieker release 1.3, which was the most recent version when performing these experiments. The JPetStore is based on the *Java Pet Store* sample application, which has originally been developed by Sun Microsystems to demonstrate the capabilities of Java EE (then referred to as J2EE). The Java Pet Store application has a long tradition of being used by researchers in experimental evaluations (e. g., by [Chen et al., 2002; Juse et al., 2003; Shams et al., 2006]).

We modified the JPetStore by *a*) adding monitoring instrumentation and *b*) adding a component to increase the CPU resource demands, as detailed below:

- *Instrumentation.* The JPetStore includes a Kieker-based instrumentation for monitoring operation executions, CPU utilization, and memory usage (Chapter 7). For monitoring operation executions, we employ the AspectJ-based (load-time weaving) and Servlet-based probes. We instrumented all 40 non-trivial operations (i. e., excluded getter and setter methods) for instrumentation by annotation (`@OperationExecutionMonitoringProbe`). The Servlet probe is used to monitor executions of the application entry point. A second Servlet probe is used to activate the monitoring of CPU and memory usage in intervals of 15 seconds. As detailed in Chapter 7, the respective probes produce the monitoring records of type `OperationExecutionRecord`, `CPUUtilizationRecord`, and `MemSwapUsageRecord`. Depending on the two experiment scenarios (Section 13.2.2), Kieker is configured to send monitoring records either to the file system or to the JMS server.
- *Additional CPU Demands.* In order to increase the complexity of the application in terms of the CPU demands, we added an additional component, called `ComplexityService`. It provides a single method that, when executed, performs a CPU-intensive computation based on a sequence of

trigonometric functions applied to a random number. Without concurrent executions, this computation takes one millisecond on the experiment infrastructure. This method is invoked once by each service of the JPetStore components `CatalogService`, `OrderService`, `AccountService` at the beginning of their executions.

Load Balancing In order to distribute the HTTP requests by the workload generators across the active JPetStore instances, we developed a basic load balancer. It maintains a list of host names referring to active application instances. Via a web-based UI, host names can be created and removed, which is done by the SLAstic framework when creating and removing respective application instances. For each new emulated user, the workload generator queries the workload generator for a host name, which is randomly selected based on the uniform distribution among the registered host names. The load balancer is part of the SLAstic framework and implemented based on Java Servlet technology.

Workload Generation For the workload generation, we use the established Apache JMeter tool, including our Markov4JMeter extension for probabilistic and intensity-varying workload [van Hoorn et al., 2008]. With respect to the probabilistic navigation profile, we use the workload model for the JPetStore that has been described in our publication on Markov4JMeter [van Hoorn et al., 2008]. We slightly modified the workload model so that at the beginning of a new user session the host name is queried from the load balancer. Details on the workload intensity will be provided in the following Section 13.2.2.

13.2.2 Workload Curve and Scenarios

In this experiment, we make the assumption that we have a good understanding of the correlation between application-level workload intensity—in this case, the number of requests to a software component per minute—and the CPU utilization. For each software component, we define a rule set specifying the number of component instances to be provided at certain workload intensity levels, e. g., five instances in periods with a workload

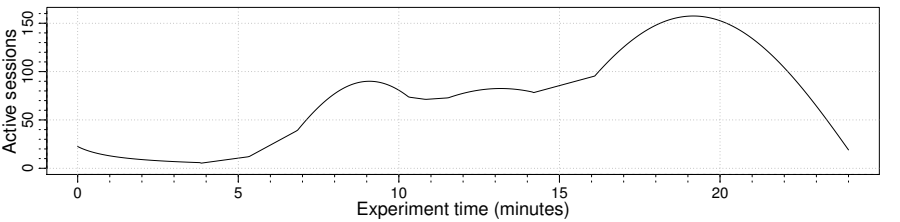


Figure 13.2. Varying workload intensity specification for the experiment

intensity of 27,000 requests per minute. Deviations between the number of component instances specified in the rule set and the number of instances actually allocated, trigger the SLAstic framework's Adaptation Planner to create an adaptation plan with the goal to achieve the requested architectural configuration. This plan is then sent to the Reconfiguration Manager for execution. This section details the workload intensity curve and the two experiment scenarios.

Workload Intensity Curve

Figure 13.2 depicts the workload intensity function used in the experiment. It specifies the number of concurrent user sessions over time to be emulated by the workload generator.

The workload intensity function mimics a workload intensity pattern that is representative for many web-based software systems (cf. Section 12.6.1): the intensity increases until a first peak at noon, which is followed by a second lower peak in the afternoon and a third peak in the evening; during the night, the intensity decreases considerably. We derived this function from 24-hour workload intensity data of an industrial production system that has been monitored with Kieker and scaled it on both experiment time and workload intensity dimension. With respect to time, we scaled the duration from the original 24 hours to 24 minutes experiment time, i. e., each experiment minute maps to a corresponding hour of the original data. With respect to the workload intensity, we scaled the curve to a maximum number of 158 concurrent sessions based on a preparative experiment in which we linearly increased the workload intensity to determine the adequate capacity with six JPetStore instances. This

workload intensity specification has been used by us in other experiments before (e. g., [Rohr et al., 2010]), using different scale factors.

Scenario 1 (Fixed Number of Nodes)

This scenario represents a static resource provisioning scenario, i. e., a fixed number of Eucalyptus instances—each hosting a JPetStore instance—is serving the requests from the workload generator throughout the entire experiment. Based on an already-mentioned preparative experiment, we determined a number of six instances to provide an adequate capacity for the configured workload.

A shell script is used to create the six Eucalyptus instances and registering them in the load balancer before starting the scenario. Each of the six Kieker instances is configured to write the monitoring data to the local file system. Hence, the SLAstic-based Adaptation Controller is not used *during* the execution of this scenario. However, after having collected the resulting monitoring logs from the six instances, they are processed by the framework in an offline mode (cf. Section 8.6.2).

Scenario 2 (Varying Number of Nodes)

This scenario represents an adaptive online capacity scenario, i. e., the number of Eucalyptus instances is changed dynamically based on the workload intensity (Figure 13.6a). We configured a rule-based adaptation strategy based on the arrival rate of requests to the JPetStore's entry level operation (assembly-level). Based on preparative experiments, we determined baseline bounds with steps of 5,000 requests. For instance, two instances provide an adequate capacity at a workload intensity level of 5,000 requests; three instances, provide an adequate capacity at a workload intensity level of 10,000 requests; etc.. Table 13.1 lists the rules for determining the number of instances to be provisioned based on the workload intensity. So far, only the *center* entry is relevant. Further details on the adaptation strategy (including also the *lower* and *upper* values) and its implementation will be provided in Section 13.3.3.

Just before starting the experiment, an initial application instance is started and registered with the load balancer. SLAstic is configured to obtain

Table 13.1. Baselines used for rule-based adaptation planning

Workload intensity bounds			Instance
Lower	**Center**	**Upper**	**count**
0	0	200	1
3,000	5,000	7,000	2
8,000	10,000	12,000	3
13,000	15,000	17,000	4
18,000	20,000	22,000	5
23,000	25,000	27,000	6

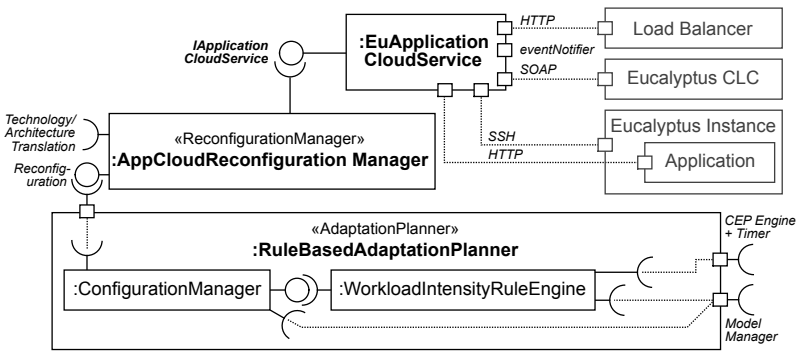


Figure 13.3. SLAstic framework extensions for the lab experiment

monitoring records via JMS and to automatically allocate or deallocate JPetStore instances based on the afore-mentioned rules.

13.3 Framework Extensions

In order to integrate the SLAstic framework with Eucalyptus cloud infrastructures and to realize a rule-based adaptation strategy, we developed three components: *a*) a cloud API with a Eucalyptus-specific implementation, *b*) a Reconfiguration Manager, and *c*) an Adaptation Planner. Figure 13.3 depicts these components (bold font style) and their integration. Referring to Figure 13.1, these components are part of the SLAstic instance. They will

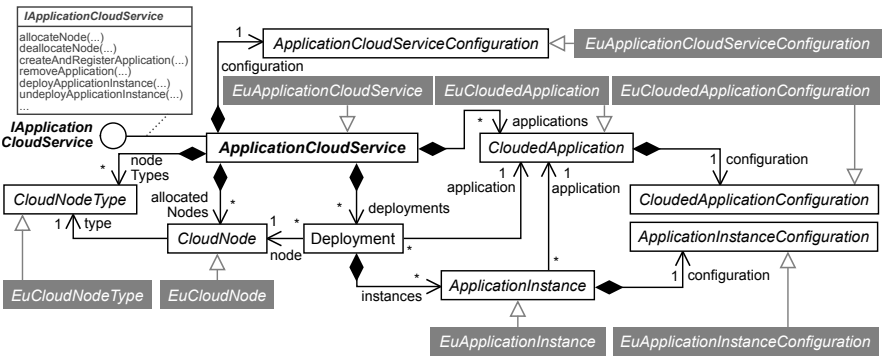


Figure 13.4. Cloud API and Eucalyptus-specific implementation (gray classes)

be described in the following Sections 13.3.1 to 13.3.3 and are available as part of the SLAstic framework [van Hoorn, 2014].

13.3.1 Cloud API and Eucalyptus-specific Implementation

Instead of integrating the SLAstic framework directly with the Eucalyptus cloud interfaces, we decided to develop an abstraction layer, which aims to ease the integration with other IaaS platforms similar to Eucalyptus. This abstraction layer comprises a vendor-independent API in form of a service interface (`IApplicationCloudService`) based on a set of—mainly abstract—data classes, as well as an abstract implementation of the interface (`ApplicationCloudService`). In order to use this API to integrate with the Eucalyptus platform, we developed a concrete Eucalyptus-specific instance of this API. The vendor-independent API and the Eucalyptus-specific extensions are depicted in Figure 13.4.

The meta-model underlying the API (Figure 13.4) allows to represent IaaS-based environments used to deploy multiple instances of application services. The environment provides a set of node types (e. g., virtual machine types and images) and corresponding nodes instances (e. g., allocated virtual machines), as well as application services and corresponding instances deployed to respective node instances. The interface provides a set of operations that serve to communicate with the respective cloud platform

and other technical services, e. g., load balancers and Application Servers. Example operations include the instantiation or termination of virtual machines, as well as the instantiation of application instances. The abstract implementation of the interface already includes abstract implementations of these operations, which mainly serve to maintain the instance of the underlying meta-model. Vendor-specific implementations of the API, e. g., for Eucalyptus or AWS, must add the logic to communicate with the respective cloud platform.

As mentioned before, we implemented a version of this API that serves to integrate with Eucalyptus but also with the remaining parts of our technical experiment infrastructure (Section 13.2.1), including the load balancer and the Tomcat server. To give an example of required extensions to the meta-model, the Eucalyptus-specific class for a node type (`EuCloudNodeType`) includes the EMI identifier; the class for a node instance includes an IP address, a host name, and the instance identifier as returned by the Eucalyptus Cloud Controller (CLC) after instantiation. As an example of Eucalyptus-specific implementations of a service operation, the allocation of a node involves *a*) a Eucalyptus call to create a virtual machine (`euca-run-instances`), *b*) waiting for the machine to become available, *c*) setting and fetching a host name via SSH, and *d*) updating the internal data model. As a second example, the deployment of an additional application instance involves *a*) the deployment of the application artifact via SCP, *b*) waiting for the application to become available, and *c*) adding the instance to the load balancer.

13.3.2 Reconfiguration Manager

In order to integrate the SLAstic framework with a specific platform—in this case, the Eucalyptus-based experiment infrastructure—it is required to develop a concrete Reconfiguration Manager. Such concrete Reconfiguration Manager extends the abstract Reconfiguration Manager already included in the SLAstic framework (Section 8.5)—which already handles a common part of executing reconfiguration plans, including rollbacks, changes to the SLAstic runtime model, etc.—by technology-specific functionality. As depicted in Figure 13.3, the developed Reconfiguration Manager uses the cloud API described in the previous Section 13.3.1. The basic procedure is to translate the architecture-level SLAstic reconfiguration operations (cf.

Chapter 10) executed as part of the interpretation of an adaptation plan into technology-specific actions using the technology/architecture translation and the cloud API in three steps: *a*) for each architectural model entity (e. g., `ExecutionContainer`) contained as parameter of the requested reconfiguration operation (e. g., component replication), lookup the identifier of the corresponding technical entity (e. g., `CloudNode`) using the technology/architecture translation (Section 8.2.8); *b*) looking up the concrete technical entities (e. g., `CloudNode`) based on the identifier using the cloud API; *c*) trigger the execution of the technical reconfiguration using respective calls to the cloud API (e. g., `deployApplicationInstance`). As part of the execution, the Reconfiguration Manager updates the SLAstic runtime model.

13.3.3 Adaptation Planner

We developed an Adaptation Planner that implements the adaptive capacity management strategy described in Section 13.2.2 (Scenario 2). As depicted in Figure 13.3, the Adaptation Planner's core components are a rule engine and a configuration manager. A rule set defines the number of Deployment Components to provision for an Assembly Component with respect to given workload intensity levels. The workload intensity refers to the number of calls to operations provided by an Assembly Component per time interval—regardless of which Deployment Component services the request. The rule engine regularly receives workload intensity measurements and evaluates the rule sets. In case a change of the number of Deployment Components is needed, the connected configuration manager is triggered to increase or decrease the number of Deployment Components for the affected Assembly Component. The configuration manager is connected to a Reconfiguration Manager—in this case the cloud Reconfiguration Manager (Section 13.3.2)—which receives and executes the adaptation plans produced by the Adaptation Planner. Note that the used monitoring events are a result of the transformation described in Section 9.2.

A rule set exists for each Assembly Component whose capacity is to be controlled at runtime. Each rule set includes a function *nextBaseline*: $(x, b) \mapsto b'$, with x being a workload intensity level and b being a baseline tuple (u, c, l, n), consisting of upper (u), center (c), and lower (l) workload intensity levels, as well as a node count (n); Let $b' = (u', c', l', n')$. If $x \geqslant l$

```
select current_timestamp as currentTimestampMillis,
       deploymentComponent.assemblyComponent, count(*)
from DeploymentComponentOperationExecution.win:time(60 seconds)
where     deploymentComponent.assemblyComponent.packageName
           = "com.ibatis.jpetstore.web"
      and deploymentComponent.assemblyComponent.name
           = "DispatcherServlet"
group by deploymentComponent.assemblyComponent
output all every 60 seconds
```

Figure 13.5. CEP query to compute invocation counts for an AssemblyComponent

and $x \leqslant u$, then $b' = b$; else b' is the baseline with the greatest c, for which $c \leqslant x$ (i. e., a *floor* lookup on an ordered set). For each Assembly Component, the rule engine maintains the current baseline, starting with a workload intensity of 0. For an incoming workload intensity value (for an Assembly Component), the next baseline is determined using the function *nextBaseline*. The configuration used for this experiment (Scenario 2) is listed in Table 13.1.

The rule engine receives workload intensity events from the SLAstic framework's CEP engine (Section 8.4.1). Figure 13.5 shows a respective CEP query specified in EPL (page 132), which serves to obtain the number of operation executions for each Assembly Component—in this case the JPetStore's entry-level component invocations in 60 second intervals. A result for this query—determined by the three components in the `select` statement—comprises the current time, the Assembly Component, and the invocation count. The rule engine triggers the configuration manager to enforce the newly desired capacity (as detailed above).

In this lab experiment, we assume that an Assembly Component has a dedicated Execution Container. Hence, in order to increase capacity, the configuration manager creates an adaptation plan with a loop (iteration count equals the number of instances to add) of *a*) an Execution Container allocation operation, and *b*) an Assembly Component replication operation request for the given Assembly Component to the previously allocated Execution Container; likewise, for the decreasing capacity an adaptation

plan is created that comprises a loop (iteration count equals number of instances to remove) of *a*) a Deployment Component dereplication operation, and *b*) an Execution Container deallocation operation. Constraints can be configured to limit the maximum number of Execution Containers per Execution Container Type and to specify what Execution Container Type is to be allocated for which Assembly Component.

13.4 Experimental Results

Section 13.4.1 starts with an initial description of the performance results for both scenarios, including the introduction of the data visualization used in this section as well as observations common to both scenarios. Sections 13.4.2 and 13.4.3 provide a more detailed discussion of scenario-specific results. A quantitative analysis of the increase of resource efficiency gained by adaptive capacity management in this experiment follows in Section 13.4.4.

13.4.1 Initial Description of Performance Results

Figure 13.6 (on page 240) shows relevant performance measurements for the Scenarios 1 and 2.

- Figure 13.6a shows the measured workload intensities for both scenarios in terms of the request arrival rate at the application's entry point. The measured workload intensity curve for Scenario 1 is smooth and appears proportional to the workload intensity specification, which is based on the number of active user sessions (Figure 13.2). For Scenario 2, the workload intensity curve particularly differs from the other curve during experiment minutes eight to thirteen.
- Figure 13.6b shows the average response time of the application's entry point for both scenarios. For both scenarios, peak values can be observed at the experiment start time. For Scenario 2, peak values can also be observed at five other points in time.
- Figures 13.6c and 13.6d show the average CPU utilization measurements and the number of allocated nodes throughout the experiment time. The average CPU utilization has been calculated by adding the individual

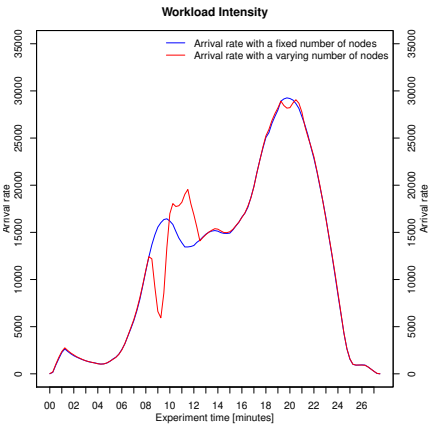
CPU utilization values for the allocated nodes and dividing this sum by the number of allocated nodes. For Scenario 1, the number of allocated server nodes is at a constant number of six throughout the experiment (Figure 13.6c). For Scenario 2, the number varies between one and six based on the adaptive capacity management (Figure 13.6d). The CPU utilization statistics for Scenario 2 show higher values than the statistics for Scenario 1, e. g., mean 0.21 vs. 0.10, maximum 0.64 vs. 0.33, and standard deviation 0.14 vs. 0.068.

Moreover, Figures 13.7 and 13.8 (pages 241 and 243) show deployment component dependency graphs extracted from the monitoring data of Scenario 1 and 2 respectively.

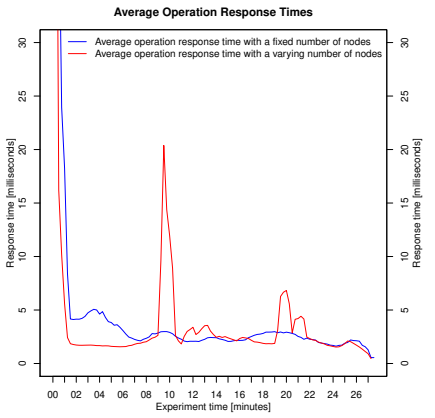
Due to an error in the experiment setup of Scenario 2, the deallocation operations were not successfully executed. Hence, starting from experiment minute 23, the number of nodes stays at a fix number of six nodes. In our opinion the impact of this error on the results described is negligible. For the further analysis in this chapter, we rescaled the CPU measurements reflecting a proportional approximation of the CPU utilization for the desired number of servers based on the CPU utilization of the actually allocated servers. These values are included in Figure 13.6d. The original measurements for the affected period correspond to the ones for Scenario 1 (Figure 13.6c). The measurements for the arrival rates are not affected. The response time measurements are the actually obtained measurements based on the erroneous configuration. The same holds for the information about the weighted calling dependencies depicted in Figure 13.8.

13.4.2 Scenario 1

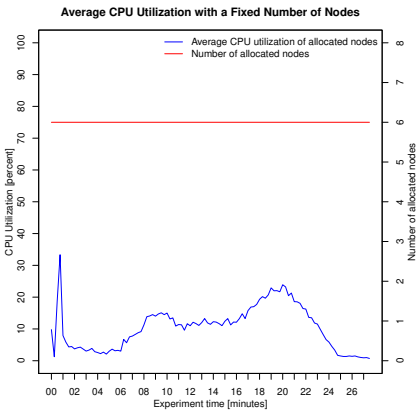
As mentioned before, both the CPU utilization and the response times in Scenario 1 show peak measurements at the beginning of the experiment. This effect is typical for Java applications when classes are used for the first time due to initial JVM operations, such as class loading. These initial operations cause both high CPU utilization and high response times. In addition to the initial peaks, both response times and CPU utilization show varying values that can be explained by the varying workload intensity the system is exposed to. However, the response times during experiment minutes two and six are unexpectedly high given the low workload intensity



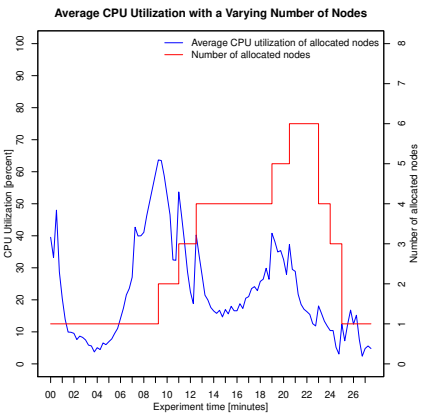
(a) Arrival rates (Scenarios 1 and 2)



(b) Average response times (Scenarios 1 and 2)



(c) Average CPU utilization (Scenario 1)



(d) Average CPU utilization (Scenario 2)

Figure 13.6. Selected performance measurements (Scenarios 1 and 2) [Fittkau, 2011]

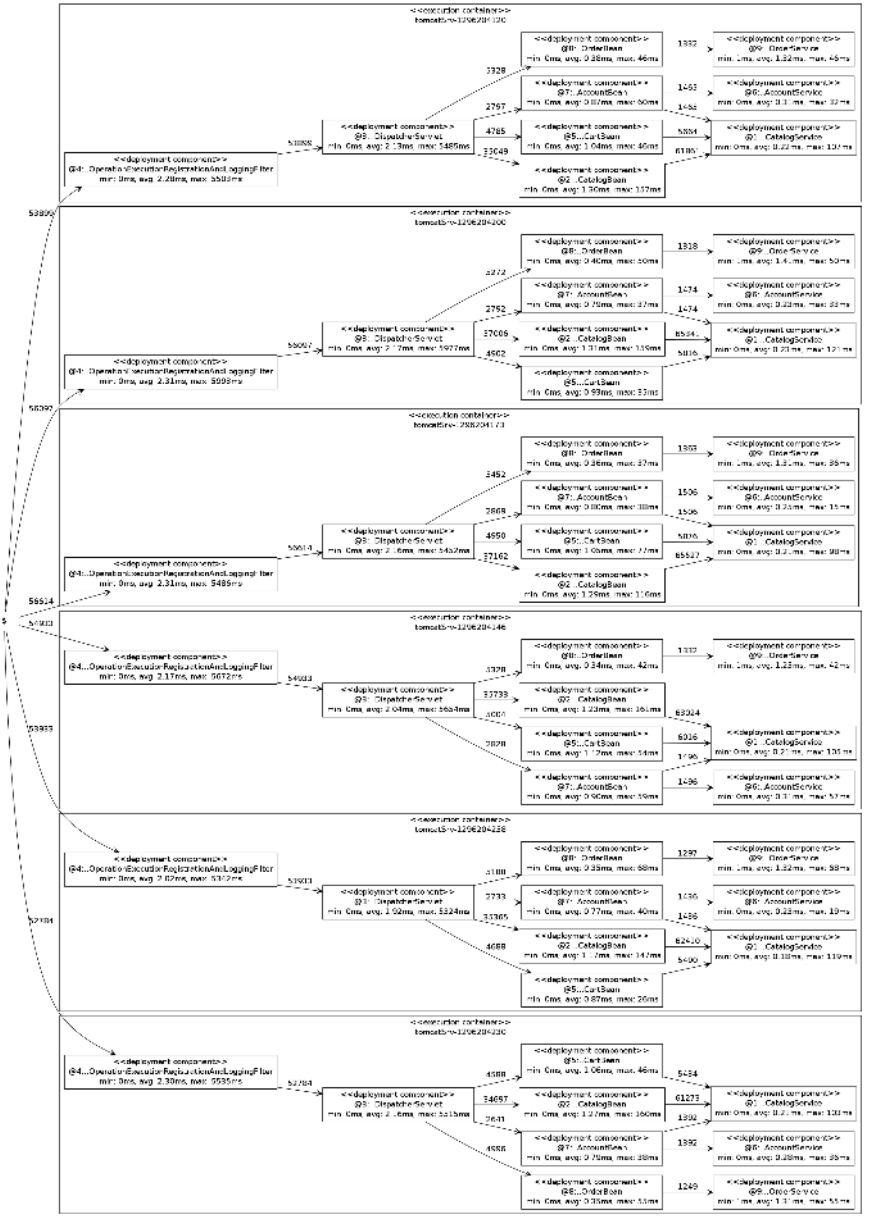


Figure 13.7. Deployment Component dependency graph (Scenario 1)

during this time period. After the second experiment minute, the response times are constantly below five milliseconds.

13.4.3 Scenario 2

Figure 13.6 includes the five points in time when additional application instances are added to the system configuration, by executing the allocation and replication runtime reconfiguration operations. Remind that the number of application instances is determined by the adaptation planner, employing a rule-based strategy based on the measured workload intensity. At least the impact of the changed configurations on the response times (Figure 13.6b) is obvious in that peaks in the response time measurements occur at the time of executing the reconfiguration operations. This effect can again be explained by the initial calls to the newly allocated instances. Also, peak values can be observed for the CPU utilization at these points in time—even tough, these peaks are not that distinct. Due to the presentation of average values, individual peaks have less impact when more instances are already allocated.

Even though the same workload intensity curve is used in both scenarios, the curves in Figure 13.6a particularly differ between experiment minutes eight and thirteen. During this period, the first additional applications instances are allocated. Compared to the arrival rate in Scenario 1, The curve indicates that a number of arrivals occurring between experiment eight and nine in Scenario 1 are shifted by one minute. This can be confirmed by the information that the total number of arrivals is almost equal between both scenarios. The total number of processed requests (corresponding to traces) was slightly higher for Scenario 2 (328,430 vs. 328,260, i. e., $\approx +0.05\%$). As a consequence, the same holds for the number of processed Kieker monitoring records (1,751,412 vs. 1,750,855, i. e., $\approx +0.03\%$) and for the number of monitored operation executions (1,750,638 vs. 1,749,414, i. e., $\approx +0.07\%$). In this experiment, a closed (session-based) workload is used. Hence, this actual workload intensity is the result of both the workload specification used by the workload generator (Section 13.2.2) and by the performance properties of the application, which (the latter) vary between the scenarios due to the different capacities. As discussed before, right after an executed reconfiguration, the performance properties like response times need some time to stabilize after initial requests.

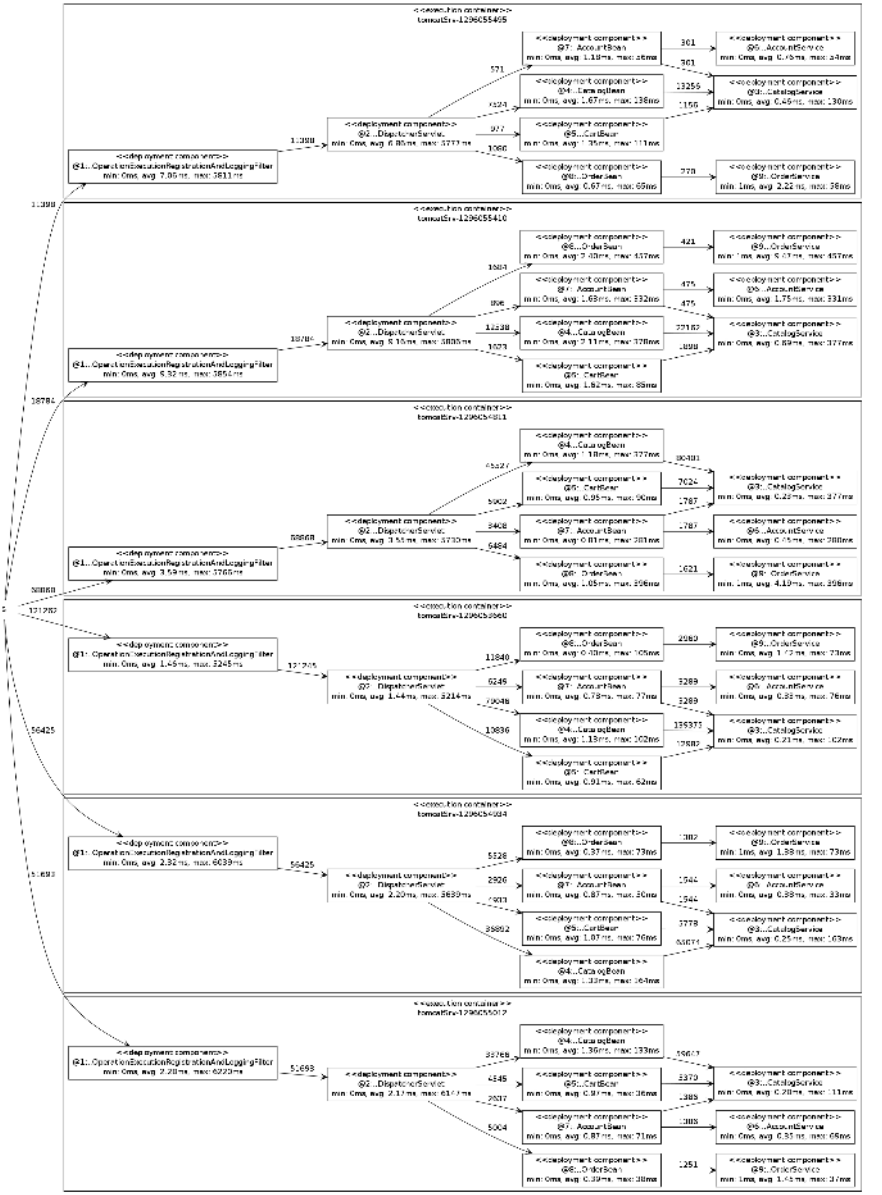


Figure 13.8. Deployment Component dependency graph (Scenario 2)

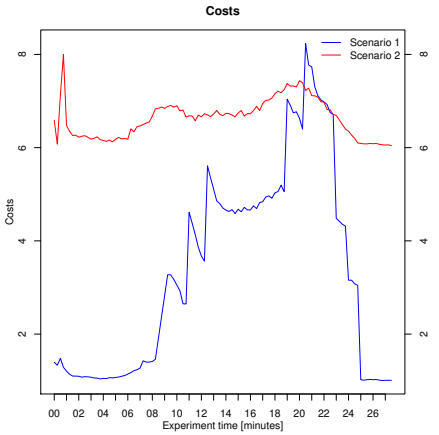


Figure 13.9. Costs for Scenarios 1 and 2

13.4.4 Quantification of Increased Efficiency

For this experiment, we were also interested in quantifying to what degree the adaptive capacity management increases resource efficiency, e. g., w.r.t. power. A common rule of thumb for power consumption of enterprise servers is that the power consumption of an idle server is 50% of the maximum amount of power it consumes when being 100% utilized, and that the power consumption is proportional to the utilization (see, e. g., [Barroso and Hölzle, 2007]). Based on this, we define the cost c_S of a server S with utilization u_S as $1 + u_S$, i. e., $c_S = 1$ if the utilization is 0.0, $c_S = 1.5$ if the utilization is 0.5, and so forth. For both scenarios, we computed the values c_S, based on the CPU utilization values and the number of allocated servers depicted in Figures 13.6c and 13.6d. Figure 13.9 depicts the resulting costs for both scenarios over the whole experiment duration. The total costs for Scenario 1 sum up to 736, while the total costs for Scenario 2 are 365. Hence, it can be concluded the adaptive capacity management in Scenario 2 saves 50% of the costs caused by the static capacity management in Scenario 1.

13.5 Summary of Results

In this experiment, we employed the SLAstic framework for architecture-based adaptive capacity management of a sample Java EE application deployed to a Eucalyptus-based IaaS infrastructure. Different extensions for the SLAstic framework have been developed for this purpose, including technology-specific implementations of SLAstic runtime reconfiguration operations, as well as a rule-based Adaptation Planner. A quantitative analysis revealed that the adaptive capacity management increased the resource efficiency by comparing two experiment scenarios under varying workload intensity—one with and a second without adaptive capacity management.

With respect to the addressed evaluation questions EQ1 (*Is the overall approach applicable to realistic scenarios?*) and EQ2 (*Does the approach have the desired properties?*) we come to the following conclusions:

- EQ1: The goal of the experiment was not to perform a systematic quantitative evaluation of the imposed monitoring overhead. However, it can be concluded that the Kieker-based monitoring did not show a perceivable perturbation (EM1.2) in both scenarios, particularly employing the distributed JMS-based monitoring log in Scenario 2. The SLAstic meta-model (EM1.3) was suitable in this approach in that it could be used as the basis for architecture-based online adaptation and no extensions were required. The SLAstic approach could successfully be integrated with the Java EE-based and IaaS-based cloud infrastructure, constituting a representative and environment for EASs (EM1.4).

- EQ2: The integration with the IaaS-based Eucalyptus infrastructure required framework extensions (EM2.1) to be developed. First, these extensions include the tailored Reconfiguration Manager with the implementation of technology-specific implementations of the selected SLAstic runtime reconfiguration operations via the developed cloud API (EM2.1.4). Second, this includes the developed rule-based Adaptation Planner. For the monitoring part (EM2.1.2), no extensions were required (EM2.1.3). As for the case study, most parts of the framework could be reused (EM2.2), particularly including the architectural modeling (EM2.2.1) and the monitoring (EM2.2.2) parts. The newly developed extensions for analysis

(EM2.2.3) and reconfiguration EM2.2.4) may be reused in further environments (cf. [Fittkau, 2012]). For this experiment, it could be shown that the runtime reconfiguration operations are suitable (EM2.3) in that they can be used to control capacity aiming for increased efficiency (EM2.3.1), and that all requests were also successful under reconfiguration (EM2.3.2). The MDSE-based techniques were successfully used in the framework to learn and update the SLAstic model of the sample application at runtime (EM2.4).

Threats to Validity External threats particularly include the representativeness of the sample application and the varying workload intensity curve used in the experiment scenarios. We are aware that the chosen application is considerably less complex than real industrial EAS. However, this is the reason why we decided to combine different types of experiments in this thesis—including the industrial case study in Chapter 12. The chosen sample application is a common (external) sample application used for performance evaluation experiments. It serves to demonstrate the applicability of our approach in a controlled lab experiment environment and builds on a representative Java EE-based technology stack. Arguing about workloads used in performance experiments has always been one of the *performance analysis rat holes* [Jain, 1991]. The workload intensity curve used in the experiment is based on the usage profile observed in a production EAS. Remind that similar curves have been observed in the workload characterization in Section 12.6.1 as well. But we are aware that the chosen workload intensity curve has a considerable impact when arguing about increased resource efficiency gained by our approach. The major internal threat in our experiment concerns the problem with failed reconfiguration operations, namely de-replication and de-allocation, at the end of the experiment, as well as the associated adjustment of CPU measurements. We already provided a discussion of the impact on the results. Moreover, the conducted reconfigurations were on a coarse-grained level, namely one application per server node. The main reason for this is that the technology-specific implementation of runtime reconfiguration requires a lot of effort. A component-level runtime reconfiguration operation for Java EE has been implemented and evaluated in the context of this thesis by Bunge [2008].

Chapter 14

Simulation-Based Evaluation

This chapter describes a simulation-based evaluation of the SLAstic approach. The SLAstic framework is used for online capacity management of an example system simulated by the SLAstic.SIM discrete-event simulator for runtime reconfigurable component-based software systems (Section 11.3).

This chapter is structured as follows. Section 14.1 details the conducted evaluation methodology. Sections 14.2 and 14.3 present the experimental setting and results. Section 14.4 provides a summary of results for the simulation-based evaluation. This chapter contains parts of a joint publication [von Massow et al., 2011] (cf. Section 5.3).

14.1 Evaluation Methodology

In this simulation-based evaluation, SLAstic.SIM and the SLAstic framework are used in three different scenarios, employing the Bookstore application that is used as a running example in this thesis. In Scenario 1, the simulated application is exposed to a constant workload intensity. A varying workload intensity curve is used in Scenarios 2—without adaptive capacity management—and in Scenario 3—with adaptive capacity management. Scenario 1 serves as a back-to-back test and basic runtime comparison with PCM's reference simulator SimuCom. Scenarios 2 and 3 serve to demonstrate the impact of a time-based adaptive capacity management employing the SLAstic framework.

The simulation-based evaluation primarily serves to address the questions EQ1 (*Is the overall approach applicable to realistic scenarios?*), EQ2 (*Does the approach have the desired properties?*), and EQ3 (*How does the approach compare to other approaches?*), as introduced in Section 5.2.5.

- With respect to EQ1, we cover the following evaluation measures:
  - EM1.3 (*Suitability of modeling language*): We evaluate whether the proposed SLAstic meta-model is able to provide an abstract view on the simulated system's architecture.
  - EM1.4 (*Suitability of separating architecture and technology*): We evaluate whether the separation of architecture and technology is suitable for the simulation-based environment.
- With respect to EQ2, we cover the following measures:
  - EM2.1 (*Extensibility of framework for specific purposes and technologies*): In order to use the Kieker and SLAstic frameworks for this experiment, extensions for monitoring (EM2.1.2), analysis (EM2.1.3), and reconfiguration (EM2.1.4) need to developed. By doing this, we evaluate the extensibility of our frameworks.
  - EM2.2 (*Reusability of framework*): We investigate to what degree existing parts of our frameworks can be reused for the simulation-based environment, particularly in terms of modeling languages (EM2.2.1), monitoring (EM2.2.2), analysis (EM2.2.3), and reconfiguration (EM2.2.4).
  - EM2.3 (*Suitability of reconfiguration operations*): Comparing the results of Scenarios 2 and 3, the impact of the reconfiguration operations on response times and CPU utilization are analyzed (EM2.3.1), and the degree of transparency is investigated (EM2.3.2).
  - EM2.4 (*Suitability of MDSE techniques*): By applying the developed MDSE techniques for model extraction to the simulated system, we evaluate their suitability.
- With respect to EQ3, we cover the following measure:
  - Scenario 1 serves to evaluate SLAstic.SIM's simulation results and the simulation time with SimuCom (EM3.2 (*Validity and performance*)).

Again, many of the measures are evaluated on a qualitative scale by demonstrating the applicability of the developed approaches based on the implemented (proof-of-concept) implementations as part of the Kieker and SLAstic frameworks.

14.2 Experimental Setting

This section describes the simulated example system (Section 14.2.1), the executed workload and adaptation scenarios (Section 14.2.2), and the software and hardware environment (Section 14.2.3).

14.2.1 Bookstore Application

The example system used for the simulation is the Bookstore application that we use as a running example in this thesis (e. g., Sections 4.5.2, 7.2 and 11.1 and Chapter 9) and in other publications (e. g., [Kieker Project, 2014a]). SLAstic.SIM simulates a PCM instance of the Bookstore system, parts of which have been described in Section 4.5.2 already.

To recapture, the Bookstore application consists of three software components, namely `Bookstore`, `Catalog`, and `CRM` (customer relationship management). The `Bookstore` component provides a service `searchBook` that allows to search for books in a catalog. The `Catalog` and `CRM` components provide the services `getBook` and `getOffers` respectively. A call to the `searchBook` service results in a deterministic trace shown in the sequence diagram in Figure 7.6 (page 114).

The entities and relationships in the PCM instance for the Bookstore are described below—based on PCM's four complementary models capturing a system's performance-relevant architectural details (cf. Section 4.5.2).

- *Repository.* The PCM repository contains component types as well as corresponding interfaces for the three Bookstore components, and respective *provides* and *requires* relationships, as depicted in Figure 4.8a (page 68). The RDSEFF of `searchBook` consists of external calls to the `getBook` and `getOffers` services, followed by a resource demand of 50 CPU units (Figure 4.8b on page 68). The RDSEFF of `getOffers` consists of an external call to `getBook`, followed by a resource demand of 20 CPU units. The service `getBook` simply contains a resource demand of 15 CPU units. The total resource demand of an execution of `searchBook` is 100 CPU units.
- *System.* The Bookstore system consists of three assembly contexts—one for each repository component. The only externally provided interface is the `IBookstore` interface. Hence, the only service that is available for external requests is `searchBook`. A PCM system diagram for the Bookstore application is shown in Figure 4.9 (page 69).

- *Resource environment.* The resource environment consists of two resource containers: `Server1` and `Server2`. Each of them has a single CPU. The CPU's processing rate varies between the evaluation scenarios, as detailed in Section 14.2.2.
- *Allocation.* Initially, each assembly context is mapped to resource container `Server 2`. `Server 1` is empty, i. e., no assembly context is mapped to it. In the scenario with reconfiguration (Section 14.2.2), the allocation is changed duration the simulation.

14.2.2 Scenarios

This chapter details the three scenarios, namely *a*) constant workload intensity, and varying workload intensity *b*) without and *c*) with reconfiguration. In each scenario, the simulated performance measurements for operation executions, CPU utilization, and active users are written to a Kieker monitoring log, employing SLAstic.SIM's monitoring facility, described on page 192.

Scenario 1: Constant Workload Intensity

The goal of this scenario is to compare the duration of simulation runs executed by SLAstic.SIM and SimuCom—using the same PCM instance of the Bookstore application. Moreover, this scenario serves as a basic back-to-back test for SLAstic.SIM's validity by comparing it with the simulation results provided by SimuCom, which is the reference simulator for PCM instances (without reconfiguration).

In both cases, the simulated system is exposed to a constant workload intensity, namely an open workload of requests to the `searchBook` service with an inter-arrival time of 0.1 time units. The aim is to provide a workload which does not overload the system with an increasing number of running transactions. In order to produce a reasonable CPU utilization, the CPUs' processing rate is set to 1,000 ticks (CPU units) per simulated time unit.

- The input workload for SimuCom is specified using the PCM workload specification, whose structure corresponds to the one shown in Figure 11.6 (page 185). For SimuCom, the maximum simulation time is set to 1,000 time units.

- For SLAstic.SIM, the workload specification is given by a synthetically created Kieker file system monitoring log with 10,000 executions of the `searchBook` method with 0.1 time units between each execution. Note that Kieker is using nano time units (by default), i. e., 100,000,000 nano units in the monitoring log refer to the inter-arrival rate of 0.1 time units. Hence, the first execution starts at time unit 0.0 and the last one at time unit 999.9.

In the experiment, we only measure the duration of the simulation. Particularly, for SimuCom we exclude the time required for code generation and compilation; for SLAstic.SIM we omit the static initialization overhead.

Scenario 2: Varying Workload without Reconfiguration

This scenario demonstrates the performance simulation driven by a trace of varying workload intensity without online capacity management, i. e., without executing runtime reconfigurations.

The input workload function for this scenario resembles one week of workload—with a seasonal pattern and a peak intensity on the weekend. Figure 14.1 (on page 252) shows the request inter-arrival time function and the corresponding arrival rates as reciprocals. Note that in the diagrams in this chapter, a time unit corresponds to a second. Such workload patterns can be observed in many real-world web-based systems (cf. Section 12.6.1). In total, the workload specification comprises 68,653 requests with a total duration of 360 time units. Opposed to the previous scenario we set the CPUs' processing rate to 100,000 ticks per simulated time unit.

As for the previous scenario, the workload specification for SLAstic.SIM is a synthetic Kieker monitoring log with respective executions of the Bookstore's entry-level service `searchBook`. The monitoring log was generated using the popular workload generator Apache JMeter along with our function timer plugin developed for this purpose.[1] The plugin takes a mathematical function of time as input and writes respective timestamps to an output file. This output file is transformed into a Kieker monitoring log.

[1] JMeter Function Timer: `http://code.google.com/p/delayfunction/`

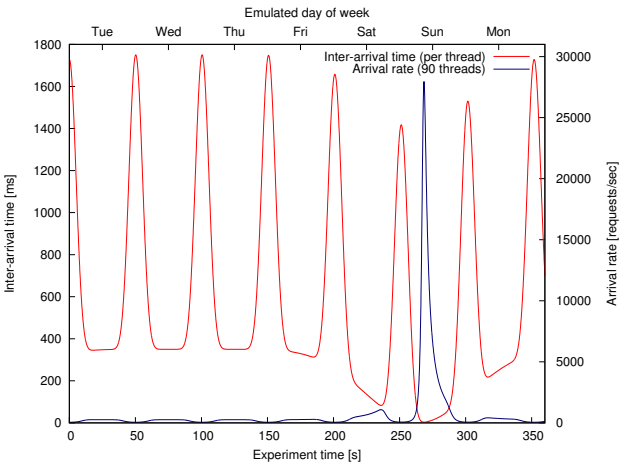


Figure 14.1. Workload intensity (Scenarios 2 and 3)

Scenario 3: Varying Workload with Reconfiguration

This scenario demonstrates the performance simulation driven by a trace of varying workload intensity and employing adaptive online capacity management controlled by the SLAstic framework. The Kieker and SLAstic (including SLAstic.SIM) frameworks are employed in the framework deployment *3. (offline/simulation)* described in Section 8.6.2 (depicted in Figure 8.6c).

The same varying workload specification as in Scenario 2, which is depicted in Figure 14.1, is used. During the time period with high workload intensity—which is supposed to be the week end—the SLAstic runtime reconfigurations from Chapter 10 are used to increase system capacity and responsiveness. We implemented SLAstic.Control components that requested the following two runtime reconfiguration plans at fixed simulation times:

1. The reconfiguration plan requested after 200 time units consists of an allocation of `Server1` followed by a subsequent replication and migration of the components `CRM` and `Catalog` respectively. Both the replication and migration have the newly allocated resource container `Server1` as its destination.

2. The inverse reconfiguration plan, requested after 300 time units, consists of the migration of component *Catalog* back to *Server2*, the de-replication of component *CRM*, and the subsequent de-allocation of *Server1*.

14.2.3 Software and Hardware Environment

All simulations are executed on a standard laptop with Ubuntu Linux. The software and hardware environment used for the experiment is listed in Table 14.1. The JVM heap space is set to 1 GB for the simulations with SLAstic.SIM and to to 2 GB for the simulations with SimuCom. The reason for devoting a larger amount of heap space to SimuCom is that it is executed inside Eclipse, which itself already requires a considerable amount of more memory.

Table 14.1. Hardware and software setup used to run the evaluation

CPU	Intel Core i5, hyper-threading enabled
RAM	4 GB
OS	Ubuntu Generic Linux kernel 2.6.32-22 SMP
Java	Sun Java Version 1.6.0_20
Heap space	1GB for SLAstic.SIM, 2GB for SimuCom 3.0

14.3 Experimental Results

The following Sections 14.3.1 to 14.3.3 describe the results of the three simulation scenarios.

14.3.1 Scenario 1: Constant Workload Intensity

Figure 14.2 shows an operation dependency graph extracted by Kieker from the monitoring log produced by SLAstic while executing Scenario 1. The included execution container name is the technical identifier of *Server2* in the PCM instance. It can be observed that the shown response time of the *searchBook* is 100 ms, which refers to 0.1 simulated time units. This is the expected response time value, having a constant workload intensity with inter-arrival rate of 0.1 time units, a CPU resource demand of 100 units, as well as a CPU processing rate of 1,000 ticks per simulated time units. Also

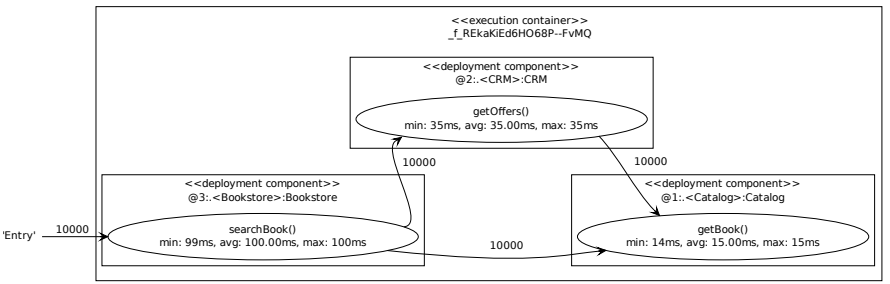


Figure 14.2. Dependency graph with response times reconstructed from Scenario 1

the simulated results of a constant CPU utilization of 100% and the constant number of a single concurrent user are as expected. These simulation results are also produced by SimuCom.

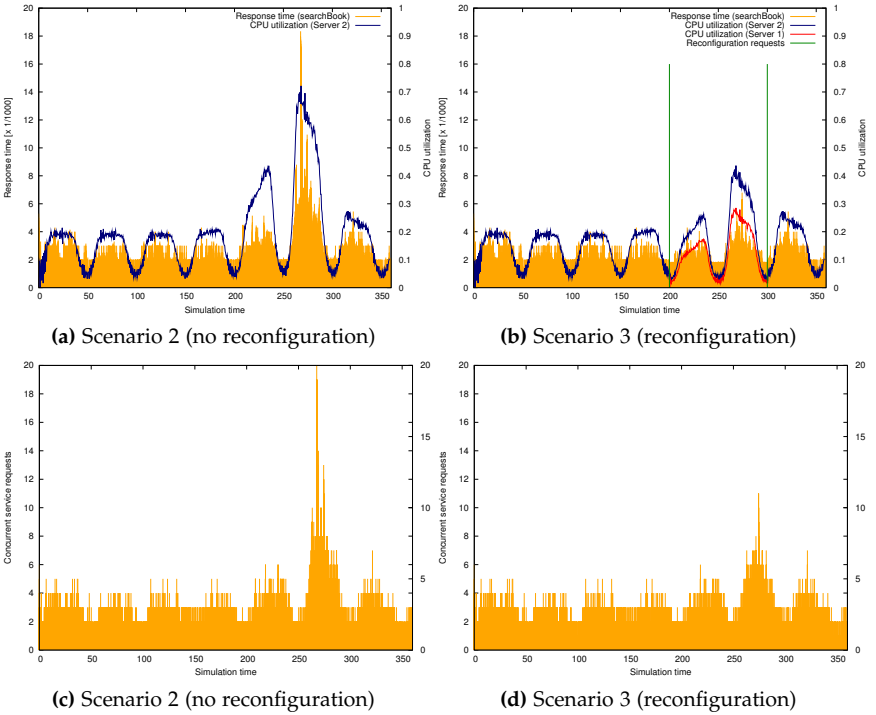
With respect to the second goal of this scenario, i. e., comparing the duration of simulation runs between SimuCom and SLAstic.SIM when executing the Bookstore model, Table 14.2 lists statistics for the duration in (milliseconds) of 50 simulation runs. Given this PCM instance, we can see that SLAstic.SIM and SimuCom are comparable regarding the overall duration of the simulation (SLAstic.SIM being slightly faster).

Table 14.2. Statistics for the duration (ms) of 50 simulation runs

	min	**median**	**mean**	**max**	**sd**
SimuCom	6,434	7,179	7,199	7,873	287.24
SLAstic.SIM	4,864	5,325	5,333	5,833	161.25

14.3.2 Scenario 2: Varying Workload w/o Reconfiguration

Figures 14.3a and 14.3c and show the simulated performance results produced by SLAstic for Scenario 2, namely response times (`searchBook` method), CPU utilization, and the number of concurrent transactions. It can be observed that the varying workload intensity highly influences these performance measures in that a seasonal pattern over the simulated days of the week can be detected. During periods with low workload intensity, the



(a) Scenario 2 (no reconfiguration)

(b) Scenario 3 (reconfiguration)

(c) Scenario 2 (no reconfiguration)

(d) Scenario 3 (reconfiguration)

Figure 14.3. Response times, CPU utilization, and number of concurrent transactions for Scenarios 2 and 3

CPU utilization is between five and ten percent, and the response times are around 0.002 simulated time units. Note that for this scenario and the following scenario, the simulated CPUs' clock rate is 100 times higher than in Scenario 1, which results in a response time of 0.001 time units for the `searchBook` methods assuming no resource contention. During periods of hight workload intensity, the number of concurrently processed transactions increases, leading to increased CPU utilization and increased service response times. A peak is reached at a simulation time of approximately 270 with a CPU load of 70% and a response time of nearly 0.018 time units.

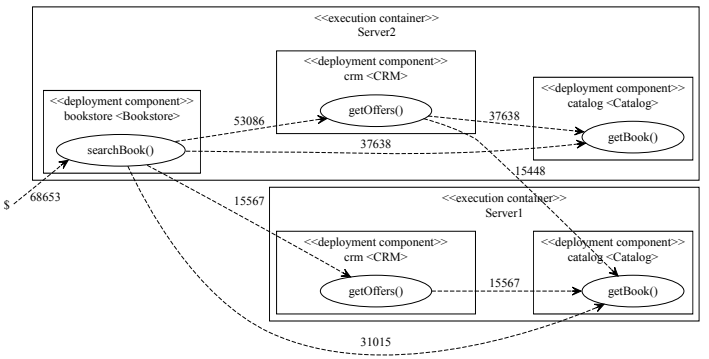


Figure 14.4. Operation dependency graph with calling frequencies (Scenario 3)

The resulting Kieker monitoring log includes 68,653 valid traces—one for each generated request. The average duration of 10 simulation runs with SLAstic.SIM was 18.6 seconds (with a standard deviation of 0.9 seconds).

14.3.3 Scenario 3: Varying Workload with Reconfiguration

Figures 14.3b and 14.3d show the simulation results for the performance measures of Scenario 3, namely response time (`searchBook` method), CPU utilization, and number of concurrent transactions. Figure 14.3b additionally includes the two points in time when the reconfiguration requests were issued. During the time period of the simulated weekend, the CPU utilization of the additionally allocated node `Server1` is shown. We can see that due to the reconfiguration both, response times and CPU utilization, can be reduced on the simulated weekend.

Figure 14.4 depicts the calling dependency graph extracted from the simulated control flow information. The diagram shows that all 68,653 requests were successfully processed, i. e., no transactions were broken. The calls to the software components and operations deployed to `Server1` result from the reconfiguration executed for the weekend. During this period, calls to the `getOffers` are distributed among the two deployment components of `CRM`. Note that the `Catalog` component is migrated to `Server1`, which explains the large number of calls to this deployment component on that server

compared to the number of calls to the `CRM` component on that server (being a replicate).

The average duration of 10 simulation runs was 18.1 seconds (with a standard deviation of 0.4 seconds).

14.4 Summary of Results

In this experiment, we employed SLAstic.SIM for simulating the Bookstore application, which is used as a running example in this thesis, in three scenarios: first, exposing it to constant workload intensity (Scenario 1); second, exposing it to varying workload intensity (Scenario 2); and finally, exposing it to varying workload intensity and controlling its capacity with the SLAstic framework (Scenario 3).

With respect to the addressed evaluation questions EQ1 (*Is the overall approach applicable to realistic scenarios?*), EQ2 (*Does the approach have the desired properties?*), and EQ4 (*How do we assess our work?*) we come to the following conclusions:

- EQ1: Even though this has not been been the main focus of this experiment and it is the running example that has been used as the application, it can be concluded that the chosen meta-model approach was suitable for this setting as well (EM1.3). Also, the intended separation of architecture and technology—in this case an application simulated with SLAstic.SIM—served useful (EM1.4).
- EQ2: Extensions were needed for the time-based Adaptation Planner (EM2.1.3). As SLAstic.SIM already provides the required components for monitoring and reconfiguration, no extension were required for these purposes (EM2.1, EM2.1.4). Hence, large parts of the remaining infrastructure could simply be reused (EM2.2–EM2.2.4). Comparing the results of Scenarios 2 and 3, it can be concluded the reconfiguration operations have the desired properties (EM2.3) w.r.t. their impact on system capacity—in this case decreasing response times during peak workload periods (EM2.3.1)—and no transactions were lost during periods of reconfiguration (EM2.3.2). The MDSE-based techniques (EM2.4) were employed for extracting the SLAstic models used at runtime.

- EQ4: The basic comparison with SimuCom in Scenario 1 showed that SLAstic.SIM simulation results are valid and that its runtime is comparable to that of SimuCom (EM3.2).

Threats to Validity The major external threat is the application used for this simulation-based evaluation, particularly w.r.t. its limited size and the PCM modeling constructs used. Hence, this evaluation serves more as a proof of concept for SLAstic.SIM and its integration with the SLAstic framework rather than as a proof about its correctness and performance. Consider that additional theoretical and experimental evaluations of SLAstic.SIM were conducted by von Massow [2010] in the context of this thesis. As for the lab experiment, the used workload intensity curve is another threat to external validity (cf. Section 13.5). Again, we aimed to choose a curve with typical characteristics of production EASs. One major threat to internal validity is the use of an old version of SimuCom.

Chapter 15

Reviewing Kieker's History, Development, and Impact

The Kieker approach and the corresponding tool support, both described in Chapter 7, form one of the key contributions of this thesis. First, Kieker serves as the basis for our SLAstic framework (Chapter 8), including model-driven instrumentation (Chapter 9), model extraction based on dynamic analysis (Chapter 9), and its experiment infrastructure (Section 8.6.2). Second, Kieker has been and is being used by ourselves and others in many academic, industrial, and collaborative contexts.

During the course of the thesis project, I played a major role in driving the Kieker project, e. g., in terms of framework design and implementation, as well as project coordination, presentation, and release management. However, it cannot be emphasized enough that many others have contributed to Kieker in various forms.

This chapter reviews the past years of Kieker development and gives some indication of the impact in terms of where and by whom Kieker has been used. Section 15.1 provides some insights into Kieker's history in terms of its origin and further evolution to the time of writing this thesis. Section 15.2 describes the development process and the tool-based development infrastructure. Section 15.3 describes contexts in which Kieker has been used by us and others.

15.1 History

This section provides a review of Kieker's history starting from its origin in 2006 to the end of 2013. We roughly divided the past years of evolution into five phases. The timeline in Figure 15.1 depicts the durations of each

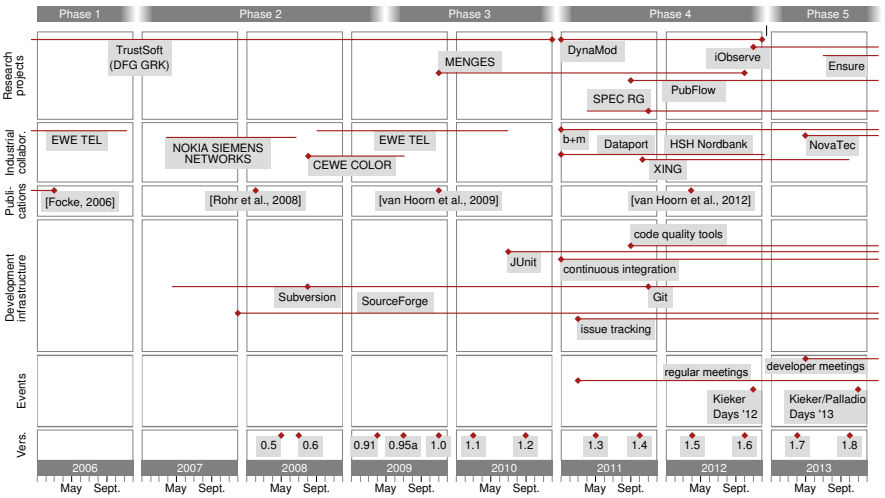


Figure 15.1. The timeline depicts durations of associated research projects, industrial collaborations, publications describing the framework, development tools, and released versions.

of these phases along with important events in the context of the Kieker project, which will be discussed in the remainder of this chapter.

15.1.1 Evolution Phases

Phase 1: 2006

Kieker originates from Focke [2006]'s Diploma thesis on performance monitoring of middleware-based applications. The thesis was conducted at the University of Oldenburg (Software Engineering Group), in collaboration with the EWE TEL GmbH, Oldenburg. As part of his work, Focke developed a performance monitoring component for Java EE applications, called *Tpmon*. Via JMX, AspectJ-based probes provided *aggregated* performance measures for Java methods: invocation counts as well as average, maximum, and minimum response times.

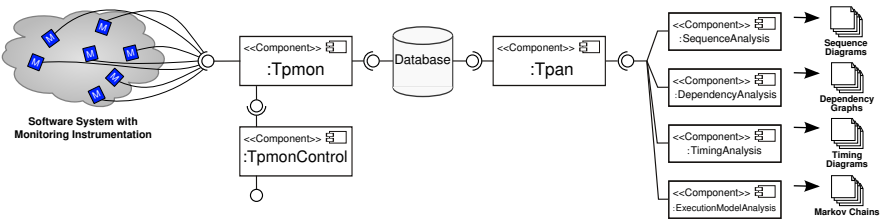


Figure 15.2. Overview of Kieker's architecture in 2007 [Rohr et al., 2008]

Phase 2: 2007–2009

Tpmon has been developed further by Matthias Rohr—who co-supervised Focke's thesis—for experimental evaluation in his PhD research on timing behavior anomaly detection [Rohr, 2014].

In the context of that research, I got in touch with *Tpmon* in 2007 during the course of my Diploma thesis [van Hoorn, 2007]: I used *Tpmon* for operation response time measurements in the experimental evaluation. At that time, *Tpmon* was tailored to measure information of operation executions (corresponding to the `OperationExecutionRecord` now included in Kieker) and log these to either the file system or a SQL database (both supporting a synchronous and an asynchronous mode).

In 2007, Kieker received its name when we prepared a first publication on the tool's architecture, and its trace extraction and visualization features [Rohr et al., 2008]. Figure 15.2 shows the visualization of Kieker's architecture from that publication. The analysis component including the trace reconstruction and visualization functionality was called *Tpan*.

We released first open-source versions of Kieker in 2008 (version 0.5 in May, version 0.6 in July).[1] These versions included only the monitoring component *Tpmon* with the afore-mentioned variants of the file system and database writers. The total number of Java classes was 12; two AspectJ-based probes were included. As part of our collaborations with CEWE COLOR and EWE TEL (detailed below), Kieker was used for monitoring production systems.

[1] A detailed overview of all releases is listed in Table 15.1 and discussed in Section 15.2.2.

In 2009, we included support for distributed tracing for Java systems that employ SOAP-based web service technology for remote communication (version 0.91). This feature was a result of our collaboration with EWE TEL as part of the case study for this thesis (Chapter 12).

During this phase, only few documentation for Kieker existed. Students needed a lot of assistance to use the tool as a basis for their work. To our knowledge, Kieker was only used by ourselves as part of our research in the DFG Graduate School on Trustworthy Software Systems (TrustSoft) and the Software Engineering Group at the University of Oldenburg.

Phase 3: 2009–2010

In 2009, we considerably restructured Kieker towards the generalized and extensible framework architecture with records, writers, readers, and analysis plugins that it has today. The restructured architecture along with results on systematic overhead benchmarks were published in our 2009 technical report [van Hoorn et al., 2009c]. Figure 15.3, showing the restructured architecture in terms of the core components and their interconnection, is taken from that report. Kieker's new architecture was released with versions 0.95a (July 2009) and 1.0 (November 2009)—the first versions containing parts of the analysis component. That year, colleagues from Kiel University (Software Engineering Group) started to join the development.

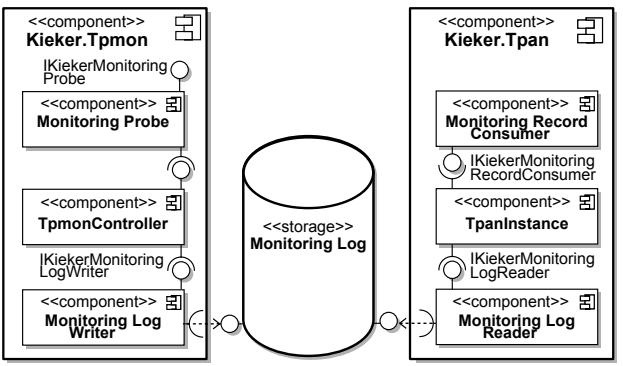


Figure 15.3. Overview of Kieker's restructured architecture [van Hoorn et al., 2009c]

In 2010, we added the system meta-model and the online trace reconstruction to the trace analysis tool. A major improvement to the documentation was made by creating the user guide with examples. *Tpmon* and *Tpan* were renamed to *Kieker.Monitoring* and *Kieker.Analysis*. We released the versions 1.1 (March) and 1.2 (September) that year.

Major results of this phase were the new framework architecture, improved documentation, and benchmarks. By the end of this phase, Kieker development moved completely to Kiel University.

Phase 4: 2011–2012

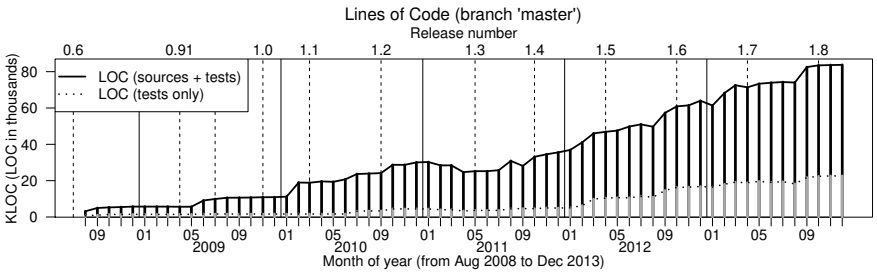
In 2011, we started to use a number of additional development tools for continuous integration, issue tracking, and improving code quality (detailed below). This was mainly driven by the successful application process for acceptance in the SPEC RG's repository of peer-reviewed tools for quantitative system evaluation and analysis—one of the core results of this phase. Version 1.3 (released in May 2011), the initial version submitted to the SPEC RG, included many new features. In version 1.4 (October 2011), which got accepted by the SPEC RG, the code quality was improved considerably based on the afore-mentioned development tool support.

In 2012, we reworked Kieker's pipes-and-filters framework, introduced event-based tracing, and released a first version of the web-based UI for configuring and executing analysis configurations. Versions 1.5 (April) and 1.6 (October) were released in this year.

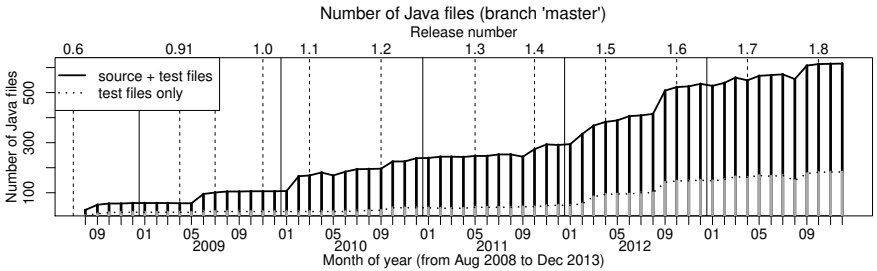
Major achievements during this phase were extensions to the feature set (including monitoring support for additional programming platforms), improvements to the code quality, and a number of additional case studies. The number of externals users grew (Section 15.3). In November 2012 we welcomed 50 participants from academia and industry to our first Kieker Days (KoSSE Symposium on Application Performance Management).

Phase 5: 2013–today

In this phase, the University of Stuttgart joined the Kieker development, due to my move to there. Since ad-hoc meetings in person between developers in Kiel and Stuttgart became more difficult, we scheduled weekly developer



(a) lines of code



(b) Java file count

Figure 15.4. Kieker's (a) LOC and (b) Java file count over time. The total numbers of these two metrics are closely correlated (0.995).

meetings via a web conference system to discuss technical topics. This system has since then also been used for the monthly regular meetings.

As a follow-up event of our first Kieker Days in 2012, we organized a joint Kieker/Palladio community meeting in Karlsruhe in 2013—again with around 50 participants.

15.1.2 Evolution of Code Size

Expressing Kieker's code size at the end of 2013 in two common metrics, it consists of 616 Java source files and more than 83,000 lines of codes (LOCs)

(83.0 KLOCs), including 183 test files with 22.6 KLOCs.[2] Taking the values from the end of September 2009 (right after version 1.0) as reference (81 files including 25 test classes; 9.0 KLOCs including 1.6 KLOCs of tests), it can be observed that since then, the number of files increased by a factor of 6.6 (factor 6.3 for tests); the LOCs increased by a factor of 8.2 (factor 13.1 for tests). The evolution of Kieker's code size with respect to these metrics is shown in Figure 15.4 and Table 15.1.

15.2 Development Process and Infrastructure

This sections provides an overview about the project meetings (Section 15.2.1), releases (Section 15.2.2), research and teaching context (Section 15.2.3), contributors (Section 15.2.4), and technical infrastructure (Section 15.2.5).

15.2.1 Project Meetings

Since March 2011, we are holding monthly meetings, referred to as *regular Kieker meetings*, to discuss topics in the Kieker context. The meeting agendas include items such as presentations and informal updates on preliminary or completed results, proposals for future activities, status updates from related research and teaching projects, release planning.[3] Proposal talks and defenses of Kieker-related study theses are integrated into the meetings. The meetings are open to everybody interested in the Kieker project. The list of attendees typically includes research staff from the involved research groups (particularly, PhD students), Kieker's student assistants, student working on a thesis in the Kieker contexts, as well external partners from collaborative research projects. As mentioned in the previous section, we are also holding weekly meetings among the Kieker developers since May 2013.

[2] LOCs are expressed as physical lines of code including comments etc. and were measured with the `wc` tool available on UNIX-like systems.

[3] The meeting agendas are available at `http://trac.kieker-monitoring.net/wiki/Meetings`.

Table 15.1. Published Kieker releases with the respective version numbers, release dates, numbers of class files, and a summary of major changes. A detailed list of changes is included in each release (*HISTORY* file).

		Number of source files		
Vers.	**Date**	**Functionality**	**Tests**	**Tests : functionality**
	Major changes			
1.8	2014/10/16	431	182	0.42
	Data bridge			
1.7	2013/04/17	396	164	0.41
	Extended adaptive monitoring; redesigned web site with live demo			
1.6	2012/10/17	375	147	0.40
	Adaptive monitoring; many additional unit tests; web-based UI (beta)			
1.5	2012/04/13	284	92	0.33
	Reworked pipes-and-filters framework; various new features (new writers, readers, and filters); event-based tracing; many additional unit tests.			
1.4	2011/10/14	199	44	0.22
	Integration of static analysis tools; Major improvements to sources, documentation, and examples.			
1.3	2011/05/19	205	42	0.20
	Various new features: e. g., periodic sampling, monitoring of system-level measures, support for custom time sources, improved configuration management			
1.2	2010/09/08	166	30	0.18
	Further improvements to the trace analysis (assembly/deployment views); Renaming of *Tpmon*/*Tpan* to *Kieker.Monitoring*/*Kieker.Analysis*; User guide.			
1.1	2010/03/04	141	25	0.18
	Improvements to the trace analysis: system meta-model, on-the-fly trace reconstruction, hierarchical calling dependency graphs, call trees.			
1.0	2009/11/18	81	28	0.35
	Minor changes and refactorings to version 0.95a.			
0.95a	2009/07/09	75	25	0.33
	Introduced extensible framework architecture (records, writers, readers, consumers, etc.); First release including core parts of *Tpan* (trace reconstruction, generation of dependency graphs and sequence diagrams, log replayer, etc.).			
0.91	2009/04/27	36	22	0.61
	Added support for distributed tracing based on *eoi* and *ess* information; Added probes for Spring web framework and CXF web services.			
0.6	2008/07/30	12	12	1.00
	Added (manual) tests for AspectJ-based instrumentation.			
0.5	2008/05/08	12	2	0.17
	First release of monitoring component *Tpmon* with AspectJ-based probes.			

15.2.2 Release Cycle and Release Preparation Schedule

Table 15.1 lists all published Kieker releases along with their version numbers, release dates, the number of class files (separated by source files and test files), as well as a summary of the major changes introduced with the respective releases. This section briefly describes Kieker's release cycle and the process for finalizing releases.

Towards a Six-Monthly Release Cycle

The release dates for the versions 0.5 to 1.2 were scheduled rather in an ad-hoc manner—mainly driven by new features and a report of these. As mentioned in Section 15.1, Kieker was mainly used by ourselves that time. Version 0.95a (a feature preview) was released as part of two talks on the Kieker's restructured extensible framework architecture, given at Kiel University.[4] The subsequent stable version 1.0 was released as part of the framework documentation in our 2009 technical report [van Hoorn et al., 2009c].

Starting with version 1.1, we have a release cycle with two Kieker versions per year. After version 1.3, we agreed on a fixed release cycle with a new version every six months. Along with the release of a version, the exact date of the next release is publicly announced on the web site. The only exception made to this so far, has been the release date for version 1.4, which has been released after five month in order to push the acceptance process for the SPEC RG tool repository.

Release Finalization Process

For the finalization of Kieker releases, we set up a five-phase process, which is instantiated five weeks before a scheduled release date. Table 15.2 lists these phases along with their (relative) start and end dates. The activities and results of each phase are as follows:[5]

[4] A. van Hoorn: "Continuous Monitoring, Analysis, and Visualization, of Java Software Behavior with the Kieker framework", presented at Kiel University (PhD seminar on July 06, 2009; Lecture on July 15, 2009).

[5] See `http://trac.kieker-monitoring.net/wiki/releases` for details.

- *Phase 1:* For each incomplete issue (features, bug fixes, etc.) associated with the current release milestone, a decision is made whether or not to resolve it for the upcoming release. In case the decision is to resolve it, a developer is assigned to the corresponding ticket in the ticket system. Other tickets must be moved to a future milestone.
- *Phase 2:* This is the last phase were code changes related to the implementation and completion of new features are allowed. No changes to the API and to external libraries must be made after this phase. A draft for the release notes can be created.
- *Phase 3:* The user guide is to be completed. This involves the finalization of the associated example projects and testing them on different platforms.
- *Phase 4:* This phase involves (manual) testing of the release archives available via the continuous integration system and the finalization of bug fixes.
- *Phase 5:* A branch for the release is created in the VCS. A release candidate is created and made available for thorough testing by the Kieker developers. The final release is being published and announced.

15.2.3 Research and Teaching Context

Kieker has been and is being developed in the context of different research and teaching activities. In most cases, Kieker is being employed for proof-of-concept implementations and quantitative evaluations of developed approaches. Kieker benefits from these activities in different forms

Table 15.2. Phases of the release finalization process with start and end times relative to the scheduled release date *t*.

Phase title	Start date	End date
1. Identification of issues to be resolved	t-5 weeks	t-4 weeks
2. API freeze and feature completion	t-5 weeks	t-3 weeks
3. Complete user guide and examples	t-4 weeks	t-2 weeks
4. Testing and bug fixing	t-4 weeks	t-1 week
5. Create, test, publish, and announce release	t-1 week	t

of contributions and to different degrees of extent. Typical contributions include:

- feedback with respect to documentation, framework usability and maturity, bug reports, etc.,
- new application scenarios and case studies,
- refined or newly introduced features, as well as
- resources, e. g., in terms of technical infrastructure and funding for technical and academic staff and student assistants working for related research projects.

The remainder of this section provides some insights into Kieker's research and teaching contexts. The research projects, including their start and end times (applying to completed projects), are also listed in the timeline in Figure 15.1.

Research

Initially, Kieker has been developed at the University of Oldenburg in the Software Engineering Group as part of the DFG-funded Graduate School on Trustworthy Software Systems (TrustSoft). In 2008, Kiel University's Software Engineering Group joined development along with Prof. Hasselbring's move to Kiel. In 2011, Kieker development moved to Kiel completely. Since then, a number of Kieker-related third-party projects have been and are being conducted there, e. g., DynaMod (2011–2012), PubFlow (since 2011), and iObserve (since 2012). Since 2013, the University of Stuttgart's Reliable Software Systems Group joined Kieker development along with my move to Stuttgart. Several Kieker-related PhD theses, each of it being a research project for itself, have been and are being conducted both as part of the afore-mentioned third-party projects and the basic funding from the involved universities.

Teaching

Kieker has been and is being used in different teaching courses conducted at the involved universities. Examples include student assignments and

guest presentations as part of lectures on software engineering and parallel/distributed systems, development projects of groups of students, and theses (Bachelor's, Master's, and Diploma).

15.2.4 Contributors

It needs to be emphasized that during the past years, many additional people contributed to Kieker in different ways and intensities. Note that we do not only consider contributions to source code. The group of contributors can be divided into researchers and students affiliated with the involved universities, as well as externals, i. e., members from other academic or industrial institutions. The contributing researchers are usually involved because they are working on a Kieker-related research projects (including PhD theses). Students usually contribute to Kieker as part of their work on Kieker-related study theses or their employment as student assistants. Externals usually contribute to Kieker during the course of collaborative projects (including papers).

15.2.5 Technical Infrastructure

This section provides some details on selected infrastructure components—namely version control, continuous integration, issue tracking, tests, code quality, and the use of the open-source hosting platform SourceForge.

Version Control

From the very beginning (at least with the beginning of phase 2), a version control system (VCS) was used to manage Kieker's source code and other documents. In the beginning, Subversion was used; in November 2011 we migrated the source code contained in the Subversion repository, including the history, to a Git repository. Since then, Git is used as the VCS technology of choice. A number of repositories exist in addition to the core repository containing the framework's sources, e. g., for web UI, examples, additional resources like talks and poster, as well as for archival of Kieker releases. The first commit to the core source code repository originates back to August 2008, i. e., after release 0.6.

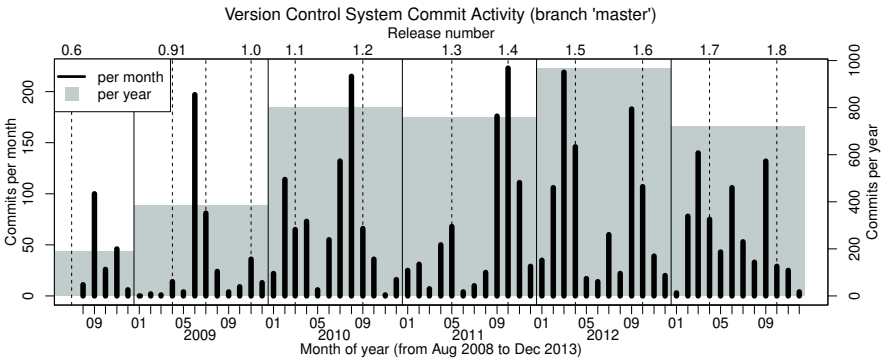


Figure 15.5. VCS activity in terms of the number of commits (per month/year)

Figure 15.5 shows the activity in Kieker's core VCS repository in terms of the number of monthly and yearly commits to the master branch. It can be observed that there has been an overall increasing trend of activity over the past years. A major yearly increase can be observed from 2009 to 2010. Within each year, the most obvious—and probably not so surprising—pattern is that peaks of activity can be observed right before a release. An observation that can be made from looking at the VCS commit messages is that they improved considerably since the migration to Git and the introduction of an issue tracking system (detailed below). Also, since the introduction of Git, new features are developed in dedicated branches, which are then merged back to the master branch as soon as the features are completed and appropriate tests exist.

Continuous Integration

In regular intervals (at least once a day but usually much more frequent), the continuous integration (CI) server retrieves Kieker's source code from the VCS, compiles it, executes the configured regression tests and static analyses, and creates and publishes snapshot versions of the release archives. The CI integrates with the unit test and static analysis tools by collecting their analysis results and visualizing them in a dashboard view. In case a build fails, e. g., due to compilation errors, failing tests, or exceeding

thresholds defined for outputs produced by the static analysis tools, the developers are informed via e-mail.

The introduction of CI for Kieker in 2011 has helped a lot to improve Kieker's product quality for several reasons, e. g., problems and their causing commits to the VCS can be pinpointed much faster; more sophisticated tests and checks have been developed since they are executed automatically and in regular intervals; the creation and basic testing of release archives has received a very high degree of automation.

Issue Tracking

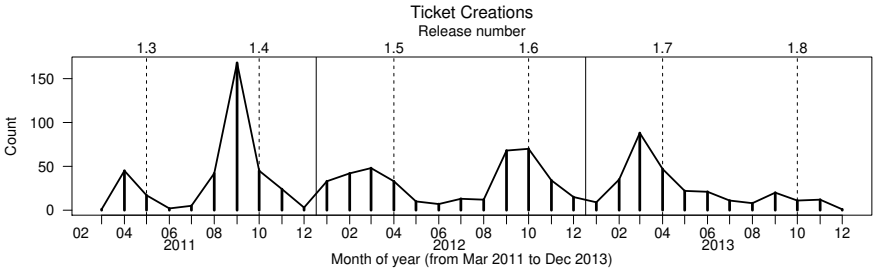
The issue tracking system provides integrated software development services like a ticket system, an interface to VCSs, and a wiki. The ticket systems allows to report issues like bugs or feature wishes, to assign them to developers, as well as to document and keep track of their progress. In addition to a web-based browser for the VCS contents, the VCS integration allows to link from tickets to VCS changesets and vice versa.

The issue tracking system has become the most important and most heavily used technical communication channel among the developers. Particularly the ticket system is used a lot by the developers to report, monitor, and discuss Kieker-related issues, including those not directly related to the software (e. g., web site and technical infrastructure). Figure 15.6 shows the activity in the issue tracking system—in terms of ticket creations and updates—since its introduction in March 2011. Peak periods of activity can be observed before a release date, e. g., for release 1.4 when we were in the acceptance process for the SPEC RG tool repository and resolved a lot of issues uncovered by the static analysis tools, which we had just introduced to the development infrastructure.

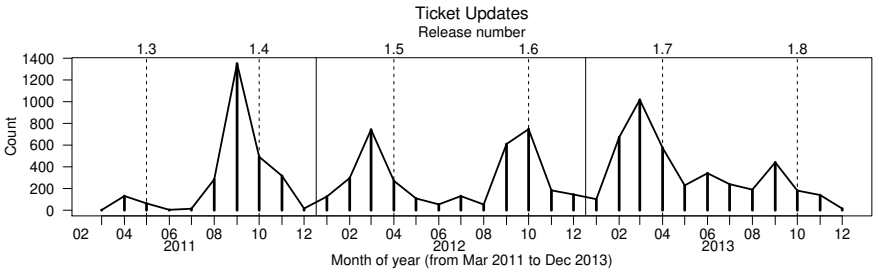
The wiki is used to publish meeting agendas and minutes, and to document reoccurring processes like setting up the infrastructure for (new) Kieker developers, planning and creating releases, etc.

Tests

For quite a long time—all versions including 1.1—no automated tests for Kieker existed. Instead, the basic functionality was tested with variants of a small Java application (namely the Bookstore being used as the running



(a) ticket creations



(b) ticket updates

Figure 15.6. Issue tracking activity in terms of (a) ticket creations and (b) ticket updates

example in this thesis), which needed to be executed manually and whose output was to be inspected manually. This kind of testing was feasible while the (small number of) developers knew every part of the code and could estimate possible impacts of changes to it. However, as the number of contributors and the amount of source code increased this approach more and more became a problem: for instance, changes accidentally had a negative impact on less frequently used functionality, which was uncovered only right before (or even after) a release when additional manual tests were performed and the cause of the problems was hard to pinpoint.

Table 15.1 (page 266) and Figure 15.4 (page 264) include the evolution of existing tests along with the amount of functionality. With the code size increase, the fraction of tests vs. functionality shows a decreasing trend

until version 1.1. Version 1.2 was the first to include tests implemented with the JUnit testing framework. Since then, the number and the quality of tests was enhanced considerably; especially with versions 1.5 and 1.6. Very much effort was put in the development of regression tests for existing features in terms of framework functionality and concrete implementations, including record (de)serialization, controllers, writers, readers, etc. As mentioned above, new features are not included in the master branch until appropriate (automatic) tests exist. Note that the JUnit tests include both unit tests and more integration-like tests.

In addition to the tests based on the JUnit framework, we developed a test script that replaced the manual inspection of the release archives. The script decompresses the release archives, inspects its contents with respect to files to be included and not to be included, and executes integration tests like executing Kieker tools from the binary release archive and comparing outputs with reference results, or builds the Kieker sources from the source release archive including the automated tests. Since the existence of this test script, problems with release archives are visible much earlier—and not close to a release date.

Code Quality

As part of the review process for the SPEC RG tool repository, we got confronted with the reviewers' results on applying static analysis tools to the Kieker source code. In 2011 (September), we integrated the three widely used static analysis tools Checkstyle, FindBugs, and PMD into our development infrastructure, i. e., build tool, IDE, and CI.

- Checkstyle focuses on the enforcement of coding conventions, e. g., formatting, naming of types and members, threshold on certain source code metrics (length of lines and classes, statement complexity, etc.), presence of license headers, and documentation. Checkstyle includes more than 130 checks at the date of this writing.
- FindBugs focuses on the detection of typical bug patterns, e. g., inconsistent overriding of methods, impossible casts, infinite loops, null pointer dereferences, concurrency issues, unused members, etc. FindBugs includes more than 400 checks at the date of this writing.

- PMD focuses on the detection of potential problems, e. g., possible bugs, as well as inefficient, overcomplicated, unused, and duplicate code. PMD includes more than 170 checks at the date of this writing.

We don't want to miss these tools for many reasons: they helped to increase the code quality, e. g., by uncovering existing and avoiding future problems, establishing and automatically enforcing project-specific coding conventions, and helping to keep the source code maintainable, which is a challenge in many-developer projects. However, we had to invest a lot of effort to come to a meaningful, project-specific configuration for each of the tools, which is particularly required for Checkstyle and PMD, and to resolve the reported problems in the source code. For instance, for PMD we excluded 40 rules. Additionally, we selectively marked a number of false alarms reported for certain locations in the Kieker source code: more than 1,100 for Checkstyle, more than 50 for FindBugs, and more than 1,200 for PMD. The tools do have some overlaps with respect to the implemented checks, but it definitely makes sense to use all of them in a complementary way.

SourceForge

In December 2007 we registered Kieker as a project at SourceForge, which is a free web-based platform for open source projects, which provides typical software development services such as VCS, issue tracking, web site, file hosting, and mailing lists. However, from these services we only use mailing lists and file hosting for publishing the Kieker release archives (including download statistics) and the JavaDoc API in HTML format.

15.3 Research and Industrial Impact

This section gives an overview about industrial collaborations (Section 15.3.1), Kieker's external use (Section 15.3.2), as well as the acceptance process for becoming a SPEC RG tool (Section 15.3.3).

15.3.1 Industrial Collaborations

During the past years, we had a number of industrial collaborations that involved the application of Kieker for dynamic analysis of production

systems and, as part of this, influenced the development of Kieker, e. g., by feature requests, feedback, and code contributions. We will briefly discuss the industrial collaborations having most impact on the evolution and evaluation of Kieker. These collaborations and case studies also served as a qualitative evaluation of the Kieker approach, e. g., concerning fine-grained continuous monitoring on application level and requirements for production scenarios, e. g., w.r.t. logging.

The afore-mentioned work by Focke [2006] initiated a collaboration with EWE TEL GmbH, Oldenburg—one of the largest regional telecommunication providers in the north of Germany. In 2008–2010, we continued this collaboration that resulted in the case study described in Chapter 12 (see also [van Hoorn et al., 2009c]), involving the instrumentation of the production Java EE-based customer web portal system with Kieker. An EWE TEL developer contributed to Kieker's distributed tracing functionality, particularly via SOAP, and integrated Kieker in the production system, where it was in use for more than half a year, as detailed in Chapter 12.

In 2008, we started a collaboration with CEWE COLOR AG & Co. OHG, Oldenburg—Europe's largest digital photo service provider. In a case study, we instrumented one front-end server node of the Java EE-based load-balanced production system—a web portal providing services, such as ordering of photo prints and other photo products (see also [Rohr et al., 2010]). A CEWE COLOR developer contributed to Kieker's Servlet- and Spring-based probes for collecting trace information, and integrated Kieker in the production system, where it was in use for one week.

As part of Bielefeld's Diploma thesis [2012], we collaborated with XING AG, Hamburg—a social network for business contacts with more than 12 million registered members as of September 2012. XING's core system, `http://xing.com`, served as a case study to evaluate the Online Performance Anomaly Detection (ΘPAD) approach developed in the thesis. ΘPAD is implemented as a Kieker analysis plugin and has been integrated in XING's monitoring architecture (see also [Bielefeld, 2012]). This collaboration continued as part of the follow-up thesis by Frotscher [2013].

In the context of the DynaMod research project [van Hoorn et al., 2011a, 2013] (01/2011–12/2012), we collaborated with the companies *a*) b+m Informatik AG, Melsdorf, *b*) Dataport AöR, Altenholz, and *c*) HSH Nordbank AG, Kiel. b+m Informatik initiated the openArchitectureWare (oAW) framework and is known for its pioneering role in developing and applying

MDE techniques and tools (also refer to Stahl and Völter [2006]). Dataport provides ICT services for the public and tax administrations of several German federal states. HSH Nordbank is a leading bank for corporate and private clients in northern Germany. The Kieker monitoring adapters for Visual Basic 6 (VB6) and .NET, which have been developed as part of the DynaMod project, were employed to analyze the case study systems AIDA-SH (Dataport) and Nordic Analytics (HSH Nordbank). For details on the latter, please refer to Magedanz [2011]. In the DynaMod context and beyond, b+m Informatik developers contributed to Kieker, e. g., in terms of functionality and bug fixes already included in recent Kieker releases. Since 2012, Kieker is integrated in b+m Informatik's generative platform b+m gear [Stahl and Völter, 2006]. Additional case studies were conducted in the context of dynamic analysis of COBOL systems [Knoche et al., 2012; Richter, 2012].

15.3.2 External Use

Kieker is not only used by us as the framework developers but also by others. Particularly in the research context, this is indicated by respective publications. Examples include the use of Kieker for research papers (e. g., Dąbrowski [2012], Markovets et al. [2013], Okanović et al. [2013], and Zheng et al. [2011]), theses (e. g., Eberlein [2011], Heger [2012], Herbst [2012] Wert [2012], and Zobel [2012]), and tools (e. g., Bartoszuk [2014], Becker et al. [2009]). Some of these works have also been conducted in collaboration with industrial partners such as SAP, Capgemini, and IBM.

15.3.3 Acceptance as SPEC RG Tool

In 2012, Kieker was one of the first two tools to pass the review process for acceptance in the SPEC RG's repository of peer-reviewed tools for quantitative system evaluation and analysis [SPEC Research Group, 2014]. Similar to the peer-reviewing process for scientific publications, submitted tools are thoroughly evaluated by a minimum number of three reviewers based on the following criteria [SPEC Research Group, 2013]: *a*) relevance to the system evaluation community, *b*) overall utility, *c*) originality or novelty, *d*) tool maturity/user base, *e*) ease-of-use and quality of documentation.

Details on the review process for the tool repository are provided in the SPEC RG's charter [SPEC Research Group, 2013].

For Kieker, the milestones of the review process were: *a*) submission of a 1-page proposal (April 2011), *b*) submission of Kieker 1.3 (June 2011) for review, *c*) the preliminary notification of acceptance with request to submit a revised version and to provide answers to the questions raised by the reviewers (September 2011), *d*) submission of Kieker 1.4 as a revised version, along with detailed comments to the reviews (October 2011), *e*) final notification of acceptance (November 2011), and *f*) public announcement on the SPEC RG web site with the actual launch of the tool repository (February 2012).

The review process and the final acceptance for the tool repository have been a great success for Kieker for several reasons, e. g., the thorough reviews from an external perspective were extremely useful as they triggered a lot of activities in the Kieker project (including the infrastructure) and helped to further improve Kieker's product quality (including quality of code and documentation); Kieker's visibility was increased considerably.

Chapter 16

Related Work

This chapter discusses work that is related to the research that has been conducted during the course of this dissertation. This discussion is structured according to the research plan that has been presented in Chapter 5. Section 16.1 describes research that is related to our overall model-driven approach for architecture-based online capacity management for component-based software systems (cf. Section 5.1). Section 16.2 discusses related architectural modeling approaches (cf. Section 5.2.1). Section 16.3 focuses on frameworks for self-adaptive software systems (cf. Section 5.2.2). Related approaches for the use of model-driven techniques for SPE (cf. Section 5.2.3) are discussed in Section 16.4. Section 16.5 focuses on related work on architecture-based online QoS management (cf. Section 5.2.4). In each section, we briefly summarize our core contributions, mention the related research areas including references to relevant literature, before discussing the relation to selected approaches. Note that the goals and a more comprehensive summary of results for each work package are provided in Section 5.2.

16.1 Overall Approach

In this thesis, we propose a model-driven approach for architecture-based online capacity management of component-based software systems (CBSSs) via runtime reconfiguration (Section 5.1). Related research areas are model-driven software engineering (MDSE), self-adaptive software systems (SASSs) employing architecture-based runtime reconfiguration, as well as model-based and measurement-based QoS evaluation (SPE). Basic concepts for these areas, including important researchers and references in these fields, have already been introduced in Chapters 2 to 4. In this section, we will

discuss selected projects and approaches that are related to our overall approach. Sections 16.2 to 16.5 focus on works specifically related to the subareas of our work, namely architectural modeling, online capacity management frameworks, model-driven techniques for SPE, and runtime reconfiguration.

- Pioneering work in the area of architecture-based and style-based self-adaptation has been conducted by three research groups, namely *a*) Kramer and Magee [1985, 1990, 2007], *b*) Oreizy et al. [1998, 2008], and *c*) Garlan et al. [2003, 2004] and Garlan and Schmerl [2004]—including other publications by these research groups on this topic. These works were the basis for the idea to follow an architecture-based approach in our work. Particularly, the work on the MAPE-K-based Rainbow framework ([Garlan et al., 2004]) includes similar concepts (cf. [Cheng, 2008]), e. g., architectural modeling and architecture-based decision making; a translation infrastructure including implementation-level monitoring probes and architecture-level gauges [ABLE group, CMU, 2009]; architecture discovery based on dynamic analysis; a language (*stitch*) to express operations, tactics, and strategies; and a customizable self-adaptation framework. Model-driven aspects are not covered in this work.
- The Descartes research group [Kounev et al., 2010], investigates techniques for self-adaptive online performance and resource management that are closely related to our approach. Like SLAstic, the Descartes project aims to provide support for the entire MAPE-K loop for self-adaptation. The Descartes has a big focus on virtualized environments, which is currently not primarily addressed by our approach. Common research topics include architectural performance modeling and the use of architectural performance models at runtime, automatic model extraction and refinement [Brosig et al., 2011], performance model transformations [Meier et al., 2011], modeling runtime reconfiguration [Huber et al., 2014], as well as online analyses such as workload forecasting and performance prediction. As mentioned in Section 5.3.2, we collaborated with this group, e. g., as part of the S/T/A approach [Huber et al., 2014] and time series analysis for online prediction of performance measures.
- Diaconescu et al. [2004; 2005; 2006] developed an approach for autonomic QoS management of component-based enterprise software sys-

tems. Based on the hypothesis that no single software component variant yields optimal quality under all environmental conditions, adaptation is performed by switching between alternate functionally equivalent component variants at runtime. The approach is embedded into their MAPE-K-based online adaptation framework AQuA (Automatic Quality Assurance). Runtime data of the managed components (e. g., response times) and the environment (e. g., workload) is continuously measured and collected employing the COMPAS monitoring framework [Mos, 2004; Mos and Murphy, 2004] (cf. Section 16.3.1), developed by the same research group. Clustering techniques are used to learn and to update the performance characteristics of the available component variants and to group these by different environmental conditions based on the monitored data. An anomaly detector identifies and signals violations of QoS requirements and relevant variations of the environmental conditions. The optimal component variant for the given environmental conditions is selected based on the learned performance information. A rule-based approach is used for anomaly detection and planning, i. e., to determine the optimal component variant and decide on its activation. The authors provide a proof-of-concept implementation for the Java EE platform and evaluated the overall approach with a sample application in lab experiments with generated workload. Note that the approach does not make use of architectural models or model-driven techniques. Also, the AQuA framework is tailored for the specific use case.

16.2 Architectural Modeling

Our core contributions with respect to the architectural modeling in WP1 (Section 5.2.1) are the SLAstic meta-model described in Chapter 6 and the integration with the Palladio Component Model (PCM) described in Chapter 11. The SLAstic meta-model allows to express relevant architectural information about a component-based software system, which is used for system instrumentation, framework initialization, and for architecture-based online analysis and runtime reconfiguration. As a result of joint work in the context of this thesis (Section 5.3), we developed the meta-model agnostic approaches MAMBA (Section 4.1.4) and S/T/A (Section 3.4.3), which are used to decorate SLAstic meta-model instances by information

relevant to QoS as well as reconfiguration and adaptation. Related work comes from the research areas of ADLs and component models, including completions [Woodside et al., 2002] for QoS-relevant information, which will be discussed below.

As mentioned in Section 3.1, comprehensive ADL classification frameworks and surveys are provided by Medvidovic and Taylor [2000] and more recently by Taylor et al. [2009]. According to Taylor et al. [2009], ADLs can be classified into *a*) first generation (no longer active) (e. g., Darwin [Magee et al., 1995], Rapide [Luckham and Vera, 1995], and and Wright [Allen and Garlan, 1997]), *b*) domain- and style-specific (e. g., Koala [van Ommering et al., 2000], Weaves [Gorlick and Razouk, 1991], and Architecture Analysis and Design Language (AADL) [Feiler et al., 2003]), *c*) as well as extensible (e. g., Acme [Garlan et al., 1997] and xADL [Dashofy et al., 2005]) languages. With respect to component models (Section 3.2), Crnković et al. [2011] provide a recent survey, including CCM [Object Management Group, Inc., 2006], COM [Box, 1998], Enterprise JavaBeans (EJB) (Section 3.3.1), OSGi [OSGi Alliance, 2012], Koala [van Ommering et al., 2000], Fractal [Bruneton et al., 2006], SOFA [Bureš et al., 2006], and Palladio Component Model (PCM) [Becker et al., 2009]. Particularly, EJB and PCM focus on EASs. Koziolek [2010] provides a survey on component performance modeling languages (Section 4.5), including approaches based on *a*) proprietary or profile-based UML extensions (e. g., building on the SPT [Object Management Group, Inc., 2005] and its successor MARTE [Object Management Group, Inc., 2011c], such as CB-SPE by Bertolino and Mirandola [2004]), *b*) and proprietary meta-models (e. g., KLAPER [Grassi et al., 2007] and PCM [Becker et al., 2009]). Selected approaches—even though not focusing on component prediction—are also presented by Cortellessa et al. [2011].

The most related work with respect to architectural modeling of CBSSs is the Palladio Component Model (PCM) [Becker et al., 2009] (see also Section 4.5.2). Originally, PCM has been developed as a model-based approach for design-time prediction of performance properties of CBSSs. PCM includes both a component model and a corresponding ADL. According to Taylor et al.'s classification, PCM can be considered a *domain*-(performance prediction) and *style*-specific (CBSSs) language. Our SLAstic meta-model builds on PCM's component model. However, we decided to develop a meta-model that is tailored to the needs of our approach. One reason was the use of SLAstic models at runtime for architecture-based

runtime reconfiguration, requiring a less detailed modeling granularity, e. g., with respect to composite components. Architectural modeling in SLAstic is pretty similar to PCM. Readers familiar with PCM will have recognized that a similar model partitioning can be found in PCM instances as well. Note that our aim was not to develop a full-blown language for architecture-based performance prediction but to model up to a level that is sufficient for architecture-based online capacity management. A bridge to the PCM-based tooling infrastructure is given by the development of the M2M transformation and decoration concept described in Chapter 11. The development of a custom meta-model and supporting tools is eased by the availability of today's MDSE technologies (Chapter 2). Both SLAstic and PCM build on the EMF-based MDSE technologies. An alternative would have been to extend PCM. So far, PCM's abilities for extensibility, e. g., using a UML-like profile mechanism, are limited. However, current efforts in this direction exist [Strittmatter et al., 2013].

Certain architectural aspects have not been integrated into the SLAstic meta-model directly. Instead, we decided to develop meta-model agnostic modeling languages in collaborations with other researchers working on similar problems. Resulting from this, we came up with the MAMBA approach that builds on the OMG's SMM [Object Management Group, Inc., 2012b] specification, and the S/T/A language. We used SMM/MAMBA to augment SLAstic models by QoS measures, e. g., as QoS properties and requirements. Related to this are QoS modeling languages (e. g., the UML QFTP profile [Object Management Group, Inc., 2008]) and SLA languages as described in Section 4.1.3 (e. g., WSLA [IBM, 2003; Keller and Ludwig, 2003], WSOL [Tosic et al., 2002; Tosic, 2004], WS-Agreement [Open Grid Forum, 2011], SLA* [Kearney et al., 2010], and SLAng [Skene, 2007; Skene et al., 2010]). Note that our goal is not to provide a new SLA language. Rather, future work includes the integration with one or more SLA language, using M2M transformations. A more detailed discussion of related work with respect to MAMBA and S/T/A are provided in our respective publications [Frey et al., 2011; Huber et al., 2014].

16.3 Online Capacity Management Framework

Our core contributions with respect to the online capacity management framework as part of WP2 (Section 5.2.2) are the Kieker and SLAstic frameworks described in Chapters 7 and 8. Kieker provides an extensible platform for creating dynamic analysis, including implementation-level instrumentation, continuous monitoring, and evaluation of application-level performance measures. Building on Kieker, the SLAstic framework provides a reusable and extensible platform for architecture-based online capacity management. Related work comes from the areas of application measurement infrastructures and self-adaptation frameworks, which will be summarized in the following two sections.

16.3.1 Application Performance Measurement

Infrastructures for performance measurement (Section 4.2), ranging from proof-of-concept research approaches to established commercial tools for production use, have been proposed throughout the past decades. Classic approaches, focusing on C/C++, include UNIX's *prof* [Bell Laboratories, 1979] and *gprof* [Graham et al., 2004] tools, for analyzing program profiles, as well as ATOM [Srivastava and Eustace, 1994] and Pin [Luk et al., 2005] for static and dynamic program instrumentation. The Application Response Management (ARM) [Johnson, 1998], maintained by The Open Group [2013], defines an API for monitoring performance information about business transactions, e. g., response times.

As mentioned in Section 4.2.2 already, performance measurement approaches for Java are typically based on the JVM-provided JVMPI/JVMTI [Oracle, 2004, 2011] infrastructure, on direct or indirect byte code manipulation, or on higher level instrumentation languages including AOP-based libraries (e. g., AspectJ [Kiczales et al., 2001]) and instrumentation DSLs (e. g., DiSL [Marek et al., 2012]). For Kieker, we make no specific assumption about the techniques and technologies used for instrumenting a system. Rather than that, it typically makes sense to use, for instance, the aforementioned approaches in a complementary way.

The COMPAS framework [Mos and Murphy, 2004] focuses on adaptive monitoring of J2EE applications. Instrumentation is conducted by adding a proxy layer around EJBs. Like Kieker, COMPAS provides extension

points for custom extensions, e. g., with respect to instrumentation and data processing. The COMPAS JEEM extension for control flow tracing was developed by Parsons et al. [2006], based on the work by Chen et al. [2002] on the Pinpoint approach. Like Kieker, COMPAS JEEM records a trace identifier, a sequence number, and call depth for observations within a trace. Magpie [Barham et al., 2003, 2004] is a tool for monitoring system-internal control flows, including requests to hardware resources, in order to extract probabilistic usage models.

Under the term application performance management (APM) [Menascé, 2002], various tools for continuously monitoring heterogeneous EAS landscapes are available. Gartner regularly analyzes the market of APM tools and publishes a report including the so-called "Magic Quadrant for Application Performance Monitoring" [Kowall and Cappelli, 2013]. As detailed by Kowall and Cappelli [2013], Gartner sees the following functional dimensions as a requirement for achieving APM objectives: *a*) end-user experience monitoring, *b*) application topology discovery and visualization, *c*) user-defined transaction profiling, *d*) application component deep-dive, and *e*) IT operations analytics. The Gartner report includes only a set of commercial tools, e. g., by companies like CA Technologies, IBM, HP, Compuware (DynaTrace), New Relic, and AppDynamics. These tools provide a rich set of features and support monitoring of system infrastructures comprises different technologies. Kieker can be seen as a platform to build an APM tool and it already includes selected APM features, e. g., with respect to discovery and visualization of distributed architectures. However, it is not the goal to compete with commercial APM tools. Kieker's strength is its flexibility and extensibility, which is usually not provided by commercial tools.

Overviews of Java profiling and monitoring tools have recently been developed in two study theses by Flaig et al. [2013], and Tel et al. [2013] respectively, who were supervised by me.

16.3.2 Self-Adaptation Frameworks

This section discusses related work on MAPE-K frameworks in research, (self-)adaptation provided by cloud infrastructures, and simulation-based performance evaluation.

MAPE-K Frameworks in Research

A number of self-adaptation frameworks and approaches have been proposed in research. Salehie and Tahvildari [2009] provide a survey comprising a taxonomy (e. g., proactive vs. reactive, model-based vs. measurement-based, level of adaptation, internal vs. external loop) as well as a description and classification of selected approaches based on the taxonomy. Note that no general-purpose framework for self-adaptation exists, as requirements and strategies for SASSs are specific to (a class of) approaches and the associated domain (see also Sections 16.1 and 16.5). However, the MAPE-K control loop [Kephart and Chess, 2003] (cf. Section 3.4.1) serves as a common blueprint for SASSs and self-adaptation frameworks. Our goal was to develop a MAPE-K-based framework for architecture-based online capacity management, for which to the best of our knowledge no suitable framework exists. Similar MAPE-K-based self-adaptation frameworks for QoS management are the aforementioned Rainbow [Garlan et al., 2003, 2004; Cheng, 2008] and AQuA [Diaconescu et al., 2004; Diaconescu and Murphy, 2005; Diaconescu, 2006] frameworks, as well as the Adaptive Server Framework (ASF) presented by Gorton et al. [2008]. Like SLAstic, Rainbow supports system adaptation based on architectural models. As opposed to this, AQuA and ASF focus on a specific technology, namely Java EE. AQuA is not designed for extensibility. ASF aims to extend existing applications by self-adaptation support.

(Self-)Adaptation Provided by Cloud Infrastructures

A major goal of cloud computing infrastructures (Section 3.3.2) is elasticity, i. e., aiming to adapt the amount of provided resources to the amount of demanded resources (cf. Definition 4.6). Therefore, today's IaaS cloud providers, such as AWS [Amazon Web Services, Inc., 2014] and Windows Azure [Microsoft, Inc., 2014], offer automatic mechanisms for adaptation as an alternative to manual control, e. g., AWS's auto scaling service. For example, virtual machines can be allocated and released based on rules defined on performance measures; load balancers are configured accordingly. First of all, we consider cloud services as technology-specific effectors in our architecture-based framework architecture, which communicate with a corresponding SLAstic Reconfiguration Manager. For Eucalyptus, this is

demonstrated in Chapter 13. With respect to the automatic mechanisms offered by the cloud providers, it could be possible to use predicted QoS measures or adaptation plans computed by the SLAstic framework as input to the rule-based adaptation.

Simulation-Based Performance Evaluation

Part of the SLAstic framework is the SLAstic discrete-even simulator SLAstic.SIM.[1] Performance evaluation of computer systems is a classical and well-studied domain for simulation [Banks, 1998; Page and Kreutzer, 2005], e. g., based on variants of queueing (network) models [Jain, 1991; Bertoli et al., 2009; Kounev et al., 2012]. For example, Java Modeling Tools (JMT) [Bertoli et al., 2009] is a tool suite for modeling and analyzing extended queueing networks. JMT includes the discrete-event simulator JSIMengine. In addition to probabilistic (multi-class) open and closed workloads, simulations can be driven by workload traces provided as log files. Like SLAstic.SIM, it is possible to use JSIMengine within external applications.

In our work, we focus on the performance simulation of software systems using performance meta-models. Simulation approaches exist for different kinds of architectural styles and corresponding models. Examples of approaches based on SPT [Object Management Group, Inc., 2005] are ArgoSPE [Gomez-Martinez and Merseguer, 2005], CB-SPE [Bertolino and Mirandola, 2004]. Cortellessa et al. [2008] proposed an approach for the simulation-based performance analysis of UML 2 models. Bause et al. [2008] proposed an approach for simulating models of service-oriented architectures (SOAs) using process chain models and the OMNeT++[2] network simulation framework.

The work most related to SLAstic.SIM is SimuCom, the simulator for PCM instances of CBSAs without runtime reconfiguration capabilities. SimuCom is integrated into the PCM modeling environment Palladio-Bench [Becker et al., 2009]. In terms of simulation correctness and simulator performance—for simulations without reconfiguration and restricted to

[1] Note that the discussion of related work on SLAstic.SIM is largely based on our SLAstic.SIM publication [von Massow et al., 2011].

[2] OMNeT++ web site: `http://www.omnetpp.org/`

the PCM modeling features supported by SLAstic.SIM—we consider SimuCom the reference implementation. Simulations with SimuCom are driven by PCM usage models of closed or open workloads, as described in Section 4.5.2. SLAstic.SIM could be easily extended to allow these kinds of workload models. In Section 14.3.1, we have used a generated workload trace equivalent to a PCM open workload usage model.

16.4 Model-Driven Online Capacity Management

Our core contributions with respect to model-driven online capacity management as part of WP3 (Section 5.2.3) are the model-driven techniques for the SLAstic framework described in Chapter 9 and the PCM integration described in Chapter 11. With respect to the results for the SLAstic framework, these comprise *a*) the model-driven generation of Kieker instrumentation, *b*) the transformation of implementation-level Kieker records into architectural SLAstic events, and *c*) the extraction and updates of SLAstic models from runtime observations. The PCM integration comprises the *a*) the transformation from SLAstic models to PCM instances and *b*) the decoration of PCM instances. Related work comes from the research areas of model-driven instrumentation and analysis, model extraction, as well as transformations between different performance modeling languages. The following Sections 16.4.1 and 16.4.2 discuss related work from the latter two areas. With respect to model-driven instrumentation and analysis, Boskovic and Hasselbring [2009] proposed the MoDePeMART, which comprises a DSL for annotating models by directives for QoS instrumentation and measures, automatic generation of measurement and measurement processing code, and the incorporation of a relational DBMS for storing measurements. As related work on architecture-based monitoring, we have already mentioned the gauge infrastructure included in the Rainbow approach (Section 16.1).

16.4.1 Model Extraction

Automatic model extraction, which can be seen as a reverse engineering and architecture discovery activity [Chikofsky and Cross, 1990; Canfora et al., 2011], is typically performed using static or dynamic analysis—or in

a hybrid form, i. e., as a combination of both. Static analysis techniques extract models from software artifacts—e. g., source or binary code—without executing them, while dynamic analysis techniques use observations from the executing software system. Note that models may also be obtained by transformations from other models, as discussed in Section 16.4.2.

Briand et al. [2006] present an approach for reverse engineering UML sequence diagrams obtained from Java systems using AspectJ. Relevant to this thesis is the extraction of architectural models including performance-relevant information, particularly based on dynamic analysis. Approaches for extracting LQNs from execution traces have been developed by Hrischuk et al. [1999] as well as Israr et al. [2007]. For PCM, model extraction approaches based on static and dynamic analysis have been proposed. Krogmann [2010] contributes the SoMoX and Beagle approaches for extracting PCM instances, including PCM's structural and behavioral views, combining static and dynamic analysis. Support for PCM extraction from sources code, based on SoMoX, is also included in the Archimetrix approach [Platenius et al., 2012]. Brosig et al. [2011] extract PCM instances from monitoring data.

16.4.2 Model Transformations

Model transformations are a core component of model-driven performance prediction approaches in SPE [Di Marco and Mirandola, 2006; Cortellessa et al., 2011]. Particularly, this concerns M2M transformations from design-level models—including performance-relevant completions [Woodside et al., 2002]—into analytical performance models (Section 4.5). Various such transformations have been proposed, e. g., from UML SPT and Use Case Maps (UCMs) to LQN [Petriu and Shen, 2002; Petriu and Woodside, 2002]. Also for PCM, which is the performance modeling languages focused on in our work, a number of transformations have been developed, e. g., from PCM to LQN [Koziolek and Reussner, 2008], from PCM to QPN [Meier et al., 2011], and from Use Case Maps (UCMs) to PCM [Vogel et al., 2013]. Note that for PCM, the reference solver for predictions is SimuCom, for which Java code is generated based on a M2T transformation.

The reason for having developed the transformation from SLAstic to PCM models was that no such transformation existed. With respect to the aforementioned existing transformations from PCM to analytic perfor-

mance models, it would be interesting to use them for online performance predictions in the future as an alternative to the simulation-based prediction.

16.5 Runtime Reconfiguration for Controlling Capacity

Our SLAstic approach employs architectural runtime reconfiguration to apply change to a controlled software system, in order to influence its QoS properties. The SLAstic meta-model and the framework provide extension mechanisms for custom reconfiguration operations. We defined and integrated five architectural reconfiguration operations with an impact on the capacity of CBSSs, namely execution container allocation and deallocation, as well as component replication, dereplication, and migration of software components.

Different types of runtime reconfiguration operations have been applied in research approaches to influence QoS properties of software systems. The following list gives selected examples:

- Software rejuvenation approaches (e. g., [Huang et al., 1995; Candea et al., 2004; Avritzer et al., 2007; Wang et al., 2007]) aim to resolve or prevent software aging effects at runtime by system or component restarts.
- Matevska [2009] investigates the redeployment of software components at runtime. An architecture-based approach is employed to optimize the point in time when to initiate a reconfiguration while minimizing system availability. The approach builds on the work by Kramer and Magee [1985, 1990] on transactional runtime reconfiguration of distributed systems and uses PCM's RDSEFF formalisms.
- As mentioned above, the aforementioned AQuA approach [Diaconescu and Murphy, 2005; Diaconescu, 2006] also employs runtime redeployment of software components to switch between alternative implementations at runtime.
- Motivated by the fact that EASs comprise a number of configuration parameters, the approach by Menascé et al. [2005] employs runtime changes of configuration parameters at runtime to impact the system's QoS.

Part IV

Conclusions & Future Work

Chapter 17

Conclusions

In this thesis, we presented our model-driven SLAstic approach for architecture-based online capacity management of component-based software systems. In addition to the overall approach, the core contributions of this thesis were made in the following categories (cf. summary of results in Chapter 5).

▻ *Architectural Modeling*

Architectural modeling is performed by a combination of the following three complementary modeling languages:

1. The SLAstic meta-model, described in Chapter 6, provides concepts to represent architectural information about distributed component-based software systems with respect to system structure, behavior, and usage, as well as adaptation.
2. For augmenting SLAstic models with quality of service (QoS) measures (Section 6.5), we employ the meta-model agnostic MAMBA/SMM approach (Section 4.1.4), which is the result of joint work in the context of this thesis.
3. For expressing reconfiguration plans (Section 6.4), we employ the meta-model agnostic meta-model approach S/T/A (Section 3.4.3), which is the result of joint work in the context of this thesis.

The results on architectural modeling provide answers to our research questions RQ1 (*Which aspects need to be modeled?*) and RQ2 (*What is a suitable modeling language?*).

▻ *Kieker Framework*

The Kieker framework, described in Chapter 7, provides an extensible and reusable platform for instrumenting, monitoring, and analyzing software systems. As detailed in Chapter 15, Kieker already gained considerable impact during the course of this thesis, by being used in other contexts of research, teaching, and industry. It became one of the first tools to be accepted for the SPEC RG's repository of peer-reviewed tools for quantitative system evaluation and analysis. It needs to be emphasized (again) that the current state of Kieker is the result of joint work with many colleagues.

The results on the Kieker framework provide answers to the research questions RQ3 (*What are relevant QoS measures to be monitored?*) and RQ5 (*What is a framework that supports the SLAstic approach?*).

▻ *SLAstic Framework*

The SLAstic framework, described in Chapter 8, provides an extensible and reusable self-adaptation platform for architecture-based online capacity management. According to the common MAPE-K control loop architecture, it is structured into components for monitoring, analysis, planning, and execution. A model manager maintains an architectural runtime model of the controlled software system. The analysis activities comprise performance evaluation, workload forecasting, performance prediction, and adaptation planning. A technology/architecture translation layer serves to abstract from concrete technologies employed by the controlled system.

This results on the SLAstic framework provide answers to the research questions RQ3 (*What are relevant QoS measures to be monitored?*), RQ4 (*What are basic analyses for online capacity management?*), and RQ5 (*What is a framework that supports the SLAstic approach?*).

▻ *Model-Driven Online Capacity Management*

In order to improve the automation of reoccurring, schematic tasks within the SLAstic approach, we developed a set of model-driven techniques to generate Kieker-based instrumentation, transform low-level monitoring data into architectural monitoring events, and to extract SLAstic models from monitoring data. These results are described in Chapter 9.

The results on this topic provide answers to the research questions RQ6 (*Where and how can MDSE techniques support the approach?*).

▻ *Runtime Reconfiguration for Controlling Capacity*

We defined and integrated a set of five architectural runtime reconfiguration operations that can be employed to control a software system's capacity at runtime. These operations are described in Chapter 10.

The results on this topic provide answers to the research question RQ7 (*What are suitable reconfiguration operations to control system capacity?*).

▻ *Integration of PCM*

Orthogonal to the previous categories, we used and integrated the Palladio Component Model (PCM) in our approach. This comprises a transformation from SLAstic models to PCM instances, a concept to decorate PCM instances by SLAstic models, a PCM-specific implementation of the previously mentioned runtime reconfiguration operations, as well as a simulator for runtime reconfigurable PCM instances.

Proof-of-concept implementations have been developed for most of the concepts proposed in this thesis. These implementations also served as the basis for an experimental evaluation in form of an industrial case study, a lab experiment, and simulations (Chapters 12 to 14). Based on defined evaluation questions and measures (quantitative and quantitative), the evaluation particularly demonstrates the applicability of the approach to realistic scenarios and the degree to which the desired properties are fulfilled.

Supplementary material for this thesis—comprising software (SLAstic, MAMBA, S/T/A), (meta-)models, examples, data, etc.—is publicly available online [van Hoorn, 2014].

Chapter 18

Future Work

This chapter outlines possible directions for future work—again, grouped according to the structure of this thesis, i. e., architectural modeling (Section 18.1), online capacity management framework (Section 18.2), model-driven online capacity management (Section 18.3), runtime reconfiguration for controlling capacity (Section 18.4), as well as the cross-cutting topic on integrating the Palladio Component Model (PCM) (Section 18.5).

18.1 Architectural Modeling

Possible future work with respect to architectural modeling includes extensions of the SLAstic meta-model and improved tool support:

- The developed SLAstic meta-model, including the S/T/A and MAMBA integration, suits the requirements with respect to the scope of this thesis. Of course, various options exist on how the meta-model could be extended for additional purposes, e. g., other quality characteristics such as reliability. Moreover, some aspects in the meta-model are modeled only very rudimentary for the sake of completeness. These aspects could be detailed in future work, e. g., with respect to more detailed modeling of connectors and network links, as well as a more powerful usage model including, for instance, time series. Reconfiguration-specific modeling could be extended to properties of reconfiguration, e. g., the time it takes for an execution container to become available.
- Currently, tool support for creating and editing instances of the SLAstic meta-model is limited to the generic tooling infrastructure provided by the Eclipse Modeling Framework (EMF). Possible future work is the development of a graphical and/or textual modeling environment for

SLAstic models, including support for the used S/T/A and MAMBA concepts. Related to this, currently no explicit textual concrete syntax or graphical concrete syntax exists—even though we used a concrete syntax similar to the one used by PCM and UML. With respect to the latter, possible future work could be the definition of a UML profile for SLAstic including the development of a bidirectional transformation between the two meta-models. Also PCM extension or customization mechanisms could be exploited—as soon as they are available in PCM (cf. [Strittmatter et al., 2013]). Both, UML and PCM extensions, would enable the visualization and editing of SLAstic models in respective tools.

18.2 Online Capacity Management Framework

Future work with respect to the online capacity management framework concerns both the Kieker and the SLAstic framework.

- The SLAstic framework developed as part of this thesis allows the integration of analysis algorithms for performance analysis, workload characterization, performance prediction, and adaptation planning. As part of the evaluation, we developed selected analyses. Possible future work includes the development and integration of additional analysis algorithms. For instance search-based software engineering approaches could be integrated to find suitable adaptation plans, tactics, forecasting algorithms, etc. The SLAstic framework including the simulation infrastructure provide a suitable experiment infrastructure. With respect to workload characterization and forecasting, future work includes the integration of the approaches by Bielefeld [2012], Frotscher [2013], and Herbst et al. [2013a].
- Cloud platforms, such as Amazon Web Services and Microsoft's Azure, include facilities for integrating custom measures for rule-based triggering of change actions to be executed by the platform, e. g., allocation of additional server instances or other cloud services. Possible future work includes the integration of the SLAstic framework in these environments.
- Both Kieker and the SLAstic are currently mainly usable via their Java APIs. An early version of a web-based UI for Kieker has been released recently [Ehmke, 2013], which will be extended in future work. The

web-based UI currently aims to be a general-purpose interface for editing and executing Kieker's pipes-and-filters configurations, as well as for displaying analysis results. Interesting future work could be to develop tailored versions of the UI, e. g., for APM. For SLAstic, a graphical UI would be desired as well, e. g., building on the aforementioned web-based KiekerUI and in combination with the desired modeling environment mentioned in Section 18.1. We have already published some ideas on this [Fittkau et al., 2014] and some first prototypes are being developed.

- During the course of this thesis, instrumentation support for selected technologies has been developed, e. g., monitoring probes for distributed tracing using AOP-like mechanisms for plain Java (based on AspectJ), Spring, CXF, etc. Possible future work includes the development of support for additional technologies. Also, the instrumentation is currently rather static in that adaptive monitoring is only possible for instrumented methods. Possible future works includes the investigation of dynamic instrumentation methods for Kieker (e. g., building on the work by Wert [2012]).

18.3 Model-Driven Online Capacity Management

The developed model-driven techniques supporting automation in the SLAstic approach focused on model-driven instrumentation, transformation of monitoring events, as well as model extraction based on dynamic analysis. Possible future work includes the development of model-driven techniques to further enhance the degree of automation and interoperability with common technologies and standards:

- CEP queries, for instance for analyzing performance measures w.r.t. SLAs, currently need to be defined manually. Possible future work includes the automatic generation of CEP queries from SLAstic models—including MAMBA annotations—to be used inside the SLAstic framework's analysis components. The same holds for the components and their configurations itself, which currently need to be defined manually. A generative approach, for instance based on state machines or another DSL, is desirable.

- Currently, SLAs are modeled directly in SLAstic models. Possible future work includes the integration with existing SLA languages (cf. Okanović et al. [2013]) including automatic M2M transformations.
- A thorough integration of the SLAstic approach into an existing generative MDSD platform would be desirable. Kieker-based model-driven instrumentation has already been integrated in an industrial architecture-centric generative platform following Stahl and Völter [2006].
- The automatic model extraction approach could be further improved, for example by integrating approaches for component discovery combining static, dynamic, and hybrid analysis (e. g., by Krogmann [2010]), as well as by providing tool support for refactoring and/or refining extracted models. The latter includes the tracking of the architecture/technology mapping.
- Measurement data is currently volatile in that it is only used as long as the data is needed to update the model and to answer registered CEP queries. Collected measurement data is currently not explicitly included in the SLAstic meta-model—even though this is already supported by SMM/MAMBA. Future work in this area mainly includes the extension of the model query mechanism, e. g., as part of the Model Manager. Preliminary work has already been conducted in a Master's project at the University of Stuttgart [Kuhn et al., 2013].

18.4 Runtime Reconfiguration

Possible future work in this area includes the integration of additional runtime reconfiguration operations and technologies, as well as additional quantitative evaluations.

- In this work, we focused on five architectural runtime reconfiguration for controlling the capacity of component-based software systems (CBSSs). These operations have been integrated into the meta-model and the framework, and technology-specific implementations have been developed, e. g., for PCM (as part of SLAstic.SIM), and Eucalyptus. Additional reconfiguration operations and technologies could be integrated as part of future work. An example reconfiguration operation is the change of

a component implementation at runtime as proposed and implemented by Matevska [2009], Bunge [2008], and Diaconescu [2006]. Moreover, runtime reconfiguration operations could be integrated based on the capabilities provided by current cloud platforms.

- The thesis includes quantitative evaluations of the impact of runtime reconfiguration on system capacity. However, a further investigation of quantitative aspects in this area could be conducted, e. g., to answer questions like *Which analysis models and solution techniques (analytic/simulation) provide feasible adaptation decisions at runtime?* or *What is a feasible time-granularity for adaptation?*

18.5 Integration of PCM

Possible future work concerning the integration of PCM includes the following topics:

- Various transformations from PCM to other performance models exist, e. g., to LQN [Koziolek and Reussner, 2008] and QPN [Meier et al., 2011]. These approaches could be exploited to transform SLAstic models into other performance models, e. g., for online performance prediction.
- The descriptions of the SLAstic2PCM transformation and the SLAstic.SIM simulator in Sections 11.1 and 11.3 include summaries of current limitations, which may serve as a basis for future work.
- Further work could be conducted in the extraction of PCM models based on Kieker and SLAstic. This includes the investigation of the validity of the extracted models.

Additional topics for future work in this area are also listed by von Massow [2010] and Günther [2011] as part of their theses.

List of Acronyms

AADL	Architecture Analysis and Design Language (originally: Avionics Architecture Description Language)
ADL	architecture description language
ADM	Architecture-Driven Modernization
AMI	Amazon Machine Image
AOP	aspect-oriented programming
API	application programming interface
APM	application performance management
ARIMA	autoregressive integrated moving average
ARM	Application Response Management
AS	Application Server
ATL	ATLAS Transformation Language
AWS	Amazon Web Services
Bash	Bourne-again shell
BNF	Backus–Naur Form
BPMN2	Business Process Model and Notation, version 2.0
CBMG	Customer Behavior Model Graph
CBSA	component-based software architecture
CBSE	component-based software engineering
CB-SPE	Component-Based Software Performance Engineering
CBSS	component-based software system
CCM	CORBA Component Model
CDN	content delivery network
CDO	Connected Data Objects
CEP	complex event processing
CEST	Central European Summer Time
CET	Central European Time
CI	continuous integration
CLI	command-line interface
CMOF	Complete MOF
COBOL	Common Business-Oriented Language

CoCoME	Common Component Modeling Example
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CPU	central processing unit
CRM	customer relationship management
CRUD	create, read, update, and delete
CSM	Core Scenario Model
CSV	comma-separated values
DBMS	database management system
DFG	Deutsche Forschungsgemeinschaft (German Research Foundation)
DNS	Domain Name System
DSL	domain-specific language
DTMC	discrete-time Markov chain
EAS	enterprise application system
EBS	Elastic Block Store
EC2	Elastic Compute Cloud
Eclipse	Eclipse IDE and RCP
Ecore	meta-meta-model included in EMF
EJB	Enterprise JavaBeans
EMF	Eclipse Modeling Framework
EMI	Eucalyptus Machine Image
EMOF	Essential MOF
EMP	Eclipse Modeling Project
EPL	Event Processing Language
ERP	enterprise resource planning
FP7	Seventh Framework Programme by the European Union
GCS	graphical concrete syntax
GPL	general purpose language
GQM	Goal Question Metric
HDD	hard disk drive
HOT	higher order transformation
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HUTN	Human-Usable Textual Notation
IaaS	infrastructure as a service
ICAC	International Conference on Autonomic Computing

ICT	information and communication technology
IDE	integrated development environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
ISO	International Organization for Standardization
IT	information technology
J2EE	Java 2 Platform, Enterprise Edition
Java	Java programming language
Java EE	Java Platform, Enterprise Edition
Java SE	Java Platform, Standard Edition
JCP	Java Community Process
JDBC	Java Database Connectivity
JMS	Java Message Service
JMT	Java Modeling Tools
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JPA	Java Java Persistence API
JRE	Java Runtime Environment
JSF	JavaServer Faces
JSP	JavaServer Pages
JSR	Java Specification Request
JTA	Java Transaction API
JVM	Java Virtual Machine
JVMPI	JVM Profiler Interface
JVMTI	JVM Tool Interface
KDM	Knowledge Discovery Meta-Model
KIT	Karlsruhe Institute of Technology
KLOC	LOCs in thousands
KM3	Kernel Meta Meta Model
KoSSE	Kompetenzverbund Software Systems Engineering
LAN	local area network
LOC	lines of code
LQN	Layered Queueing Network
M2M	model-to-model
M2T	model-to-text
MAMBA	Measurement Architecture for Model-Based Analysis

MAPE-K	Modeling, Analysis, Planning, Execution, Knowledge
MARTE	UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems
MDA	Model-Driven Architecture
MDD	model-driven development
MDE	model-driven engineering
MDSD	model-driven software development
MDSE	model-driven software engineering
MOF	Meta Object Facility
MOM	message-oriented middleware
MTTF	mean time to failure
MTTR	mean time to repair
NATO	North Atlantic Treaty Organization
NIST	National Institute of Standards and Technology
NTP	Network Time Protocol
oAW	openArchitectureWare
OCL	Object Constraint Language
OGF	Open Grid Forum
OMG	Object Management Group
ΘPAD	Online Performance Anomaly Detection
OSGi	A Java component model (originally, OSGi was an acronym for Open Services Gateway initiative)
Object-Z	Object-Z Specification Language
PaaS	platform as a service
PCM	Palladio Component Model
PN	Petri Net
QFTP	UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms
QM	Queueing Model
QN	Queueing Network
QoS	quality of service
QPN	Queueing Petri Net
QVT	Query/View/Transformation
RCP	rich client platform
RDS	Relational Database Service
RDSEFF	Resource Demanding SEFF
REST	Representational State Transfer

RMI	Remote Method Invocation
RPC	remote procedure call
RRA	Round Robin Archive
RRD	Round Robin Database
RRDtool	Round Robin Database tool
RSA	Rational Software Architect
S3	Simple Storage Service
SaaS	software as a service
SASS	self-adaptive software system
SCP	Secure Copy
SEAMS	Symposium on Software Engineering for Adaptive and Self-Managing Systems
SEFF	Service Effect Specification
SEI	Carnegie Mellon Software Engineering Institute
SLA	service level agreement
SLA*	a language for specifying SLAs
SLAng	a language for specifying SLAs
SLAstic	name of the approach developed in this thesis
SLO	service level objective
SMM	Structured Metrics Meta-Model
SOA	service-oriented architecture
SOAP	protocol for exchanging structured information in computer networks (originally: Simple Object Access Protocol)
SPE	software performance engineering
SPEC	Standard Performance Evaluation Corporation
SPEC RG	SPEC Research Group
SPEL	Software Performance Engineering Lab
SPT	UML Profile for Schedulability, Performance, and Time
SQL	Structured Query Language
SSH	Secure Shell
SVN	Subversion
SWaP	Space, Watts and Performance
T2M	text-to-model
TCO	total cost of ownership
TCP/IP	Transmission Control Protocol/Internet Protocol
TCS	textual concrete syntax

UC	University of California
UCM	Use Case Map
UI	user interface
UML	Unified Modeling Language
UML 2	UML, version 2
URL	Uniform Resource Locator (originally: Universal Resource Locator)
UTC	Coordinated Universal Time
VB6	Visual Basic 6
VCS	version control system
WCOP	Workshop on Component-Oriented Programming
WS-Agreement	Web Service Agreement
WSDL	Web Services Description Language
WSLA	Web Service Level Agreement
WSOI	Web Service Offerings Infrastructure
WSOL	Web Service Offerings Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
Xtend	a programming language on top of Xtext
Xtext	framework for development of programming languages and DSLs
Z	Z Specification Language

List of Figures

2.1 Four-layered meta-modeling stack 12
2.2 Model transformation schema 14
2.3 Equivalent representations for EMF models 18

3.1 Context and conceptual model of an architecture description according to the ISO/IEC/IEEE Standard 42010:2011(E) . . . 23
3.2 S/T/A meta-model 36

4.1 Example SLA* (template) specification 48
4.2 Core SMM meta-model concepts 50
4.3 MAMBA extension mechanism for aggregate functions as well as collective and periodic measures 53
4.4 MAMBA framework with measurement providers 54
4.5 Capacity planning methodology by Menascé and Almeida . . 59
4.6 Closed Queueing Network 66
4.7 UML SPT sequence and deployment diagrams 66
4.8 PCM repository contents of the Bookstore example application 68
4.9 PCM system diagram of the Bookstore application 69

5.1 Model-driven instrumentation and analysis in the DynaMod approach 82

6.1 Object-Z specification of the SystemModel meta-class 93
6.2 Subset of the meta-classes for the type repository 93
6.3 Object-Z specification of the ComponentType meta-class . . . 94
6.4 Object-Z specifications of meta-classes for resource types and resource specifications 96
6.5 Meta-model excerpts of the component assembly model . . . 97
6.6 Core meta-classes and relations for the execution environment model 98

6.7 Component deployment model containing deployment components . . . 98
6.8 Meta-classes for operation execution and resource usage events 99
6.9 Meta-classes and relationships for representing traces 100
6.10 Usage model 101
6.11 Reconfiguration plan including reconfiguration actions 103
6.12 Reconfiguration model including reconfiguration capabilities and properties 104
6.13 Annotations for QoS measures and instrumentation 105

7.1 Kieker's core components, assembly, and interactions 108
7.2 Abstract and example Monitoring Record meta-classes 109
7.3 Example file system representation of Monitoring Records . . 110
7.4 Core entities of the Kieker monitoring and analysis framework 111
7.5 Core classes of the meta-model for representing component-based software systems, which is used by Kieker's trace analysis 114
7.6 Tracing-related terminology in Kieker 114
7.7 Meta-model used by Kieker for representing reconstructed traces. 115
7.8 Selected visualizations generated by Kieker based on reconstructed trace information 118
7.9 Kieker's core components, extension points, and features . . . 120

8.1 Top-level views on the SLAstic framework architecture 126
8.2 Decomposition of the Model Manager into subcomponents . 128
8.3 Composite structure of the Adaptation Controller and the Analyzer 132
8.4 EPL statement to periodically collect the number of operation executions for each Assembly Component observed within the past 10 seconds every 5 seconds. 133
8.5 Activity diagram of the (proactive) analysis phase 134
8.6 Framework uses and integrations for online and offline analyses 139
8.7 Components for online analysis via JMS 141

9.1 Overview and examples of model-driven instrumentation approach 145
9.2 Transformation result for CPUUtilizationRecord 147

9.3 Transformation result for OperationExecutionRecord 149
9.4 EPL statement to collect operation executions of a trace 151
9.5 Transformation results for CPUUtilizationRecord and OperationExecutionRecord in architecture discovery mode 154

10.1 SLAstic reconfiguration operations 162
10.2 S/T/A actions corresponding to the SLAstic runtime adaptation operations 167

11.1 Diagram of the PCM repository for the Bookstore created by SLAstic2PCM 175
11.2 Diagram for an RDSEFF created by SLAstic2PCM 178
11.3 Diagram of the PCM system model for Bookstore created by SLAstic2PCM 180
11.4 PCM resource environment model created by SLAstic2PCM (Bookstore system) 182
11.5 Diagram of the PCM allocation model created by SLAstic2PCM 183
11.6 Diagram for the `pcm::UsageScenario` created by SLAstic2PCM . 185
11.7 Implementation of the SLAstic transformation SLAstic2PCM 185
11.8 Example SLAstic2PCM rule for transforming a SLAstic execution container into a PCM resource container 186
11.9 PCM decoration example for component types 188
11.10 High-level architecture and integration of SLAstic.SIM 191
11.11 SLAstic.SIM reconfiguration plan and operations 193
11.12 Activity diagram for the component de-replication operation 194

12.1 Architecture of the case study system 203
12.2 Number of invalid traces (per day) during the observation period 210
12.3 Number of valid traces and average maximum execution stack sizes during the observation period 211
12.4 Visualizations of selected architectural models reconstructed by Kieker 212
12.5 Distribution of calling frequencies from the *doFilter* operation to the *handleMessage* operation 215
12.6 Arrival rates of the Servlet entry (assembly-level) over one week 216

12.7 Arrival rates and correlations for selected assembly components 217
12.8 CPU utilization (5 minute intervals) of the four servers over one week and for over a selected day 220
12.9 Statistics and probability density functions for CPU utilization 221

13.1 Overview of the experiment infrastructure 227
13.2 Varying workload intensity specification for the experiment . 231
13.3 SLAstic framework extensions for the lab experiment 233
13.4 Cloud API and Eucalyptus-specific implementation 234
13.5 CEP query to compute invocation counts for an Assembly-Component 237
13.6 Average CPU utilization of allocated nodes 240
13.7 Deployment Component dependency graph (Scenario 1) . . . 241
13.8 Deployment Component dependency graph (Scenario 2) . . . 243
13.9 Costs for Scenarios 1 and 2 244

14.1 Workload intensity (Scenarios 2 and 3) 252
14.2 Dependency graph with response times reconstructed from Scenario 1 254
14.3 Response times, CPU utilization, and number of concurrent transactions for Scenarios 2 and 3 255
14.4 Operation dependency graph with calling frequencies (Scenario 3) 256

15.1 Kieker timeline 260
15.2 Overview of Kieker's architecture in 2007 261
15.3 Overview of Kieker's restructured architecture 262
15.4 Kieker's LOC and Java file count over time 264
15.5 VCS activity in terms of the number of commits 271
15.6 Issue tracking activity 273

List of Tables

2.1 Important OMG modeling specifications 16

3.1 Selected Java SE and Java EE technologies 30

4.1 Comparison of pure SMM and MAMBA 52

5.1 Evaluation questions, measures, methods, and scales of measurement 85

8.1 Implementation classes for conceptual framework components 138

9.1 Kieker record types and corresponding SLAstic meta-classes 146
9.2 Example results of type and operation signature name abstraction 158

10.1 Model Manager's reconfiguration operation signatures 169

11.1 High-level mapping between the SLAstic and PCM metamodel partitions 174

12.1 Configuration for each of the RRDs and the contained RRAs . 206
12.2 Assignment of values to CPUUtilizationRecord fields 206
12.3 Basic statistics about the (raw) Kieker monitoring logs with operation executions 208

13.1 Baselines used for rule-based adaptation planning 233

14.1 Hardware and software setup used to run the evaluation . . . 253
14.2 Statistics for the duration (ms) of 50 simulation runs 254

15.1 Published Kieker releases 266
15.2 Phases of the release finalization process 268

Bibliography

[ABLE group, CMU 2009] ABLE group, CMU. Dasada Gauge Infrastructure. http://www.cs.cmu.edu/~able/research/rainbow/gaugeinf.html, 2009. (cited on page 280)

[ACM SIGMETRICS 2009] ACM SIGMETRICS. *SIGMETRICS Performance Evaluation Review*, 36(4), 2009. Special issue on tools for computer performance modeling and reliability analysis. (cited on page 66)

[Allen and Garlan 1997] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997. (cited on pages 27 and 282)

[Allspaw 2008] J. Allspaw. *The Art of Capacity Planning*. O'Reilly, 2008. (cited on page 58)

[Amazon Web Services, Inc. 2014] Amazon Web Services, Inc. Amazon Web Services. http://aws.amazon.com/, 2014. (cited on pages 2, 32, and 286)

[Arlitt et al. 2001] M. F. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology (TOIT)*, 1(1):44–69, 2001. (cited on pages 1, 63, and 64)

[Armbrust et al. 2009] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009. (cited on pages 2 and 31)

[Avritzer et al. 2007] A. Avritzer, A. B. Bondi, and E. J. Weyuker. Ensuring system performance for cluster and single server systems. *Journal of Systems and Software*, 80(4):441–454, 2007. (cited on page 290)

[Balbo 2007] G. Balbo. Introduction to generalized stochastic petri nets. In *Proceedings of the 7th International School on Formal Methods for the Design of Computer, Communication, and Software System (SFM '07)*, LNCS, pages 83–131. Springer, 2007. (cited on page 65)

[Balsamo and Marin 2007] S. Balsamo and A. Marin. Queueing networks. In *Proceedings of the 7th International School on Formal Methods for the Design of Computer, Communication, and Software System (SFM '07)*, volume 4486 of *LNCS*, pages 34–82. Springer, 2007. (cited on pages 41 and 65)

[Balsamo et al. 2004] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering (TSE)*, 30(5):295–310, 2004. (cited on page 65)

[Banks 1998] J. Banks, editor. *Handbook of Simulation: Modelling, Estimation and Control*. Wiley & Sons, 1998. (cited on pages 189 and 287)

[Banks et al. 2009] J. Banks, J. S. Carson, II, and B. L. Nelson. *Discrete-Event System Simulation*. Prentice Hall, 5 edition, 2009. (cited on page 189)

[Barham et al. 2003] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HOTOS '03)*, pages 85–90. USENIX Association, 2003. (cited on page 285)

[Barham et al. 2004] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI '04)*, pages 259–272. USENIX Association, 2004. (cited on page 285)

[Barroso and Hölzle 2007] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007. (cited on pages 2 and 244)

[Bartoszuk 2014] C. Bartoszuk. Callcount project. `https://github.com/cbart/fly/tree/master/Callcount`, 2014. (cited on page 277)

[Basili et al. 1994] V. R. Basili, G. Caldiera, and H. D. Rombach. Goal Question Metric paradigm. In *Encyclopedia of Software Engineering*, volume 1, pages 528–532. John Wiley & Sons, 1994. (cited on page 84)

[Bause and Buchholz 1998] F. Bause and P. Buchholz. Queueing Petri nets with product form solution. *Elsevier Performance Evaluation*, 32(4):265–299, 1998. (cited on page 65)

[Bause and Kritzinger 2002] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets – An Introduction to the Theory*. Vieweg Verlag, 2nd edition, 2002. (cited on page 65)

[Bause et al. 2008] F. Bause, P. Buchholz, J. Kriege, and S. Vastag. A framework for simulation models of service-oriented architectures. In *Proceedingsof the SPEC International Performance Evaluation Workshop (SIPEW '08)*, volume 5119 of *LNCS*, pages 208–227. Springer, 2008. (cited on page 287)

[Becker et al. 2006a] S. Becker, L. Grunske, R. Mirandola, and S. Overhage. Performance prediction of component-based systems: A survey from an engineering perspective. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 169–192. Springer, 2006a. (cited on page 65)

[Becker et al. 2006b] S. Becker, W. Hasselbring, A. Paul, M. Boskovic, H. Koziolek, J. Ploski, A. Dhama, H. Lipskoch, M. Rohr, D. Winteler, S. Giesecke, R. Meyer, M. Swaminathan, J. Happe, M. Muhle, and T. Warns. Trustworthy software systems: A discussion of basic concepts and terminology. *SIGSOFT Software Engineering Notes (SEN)*, 31(6):1–18, 2006b. (cited on pages 1 and 39)

[Becker et al. 2009] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Elsevier Journal of Systems and Software (JSS)*, 82(1):3–22, 2009. (cited on pages 2, 3, 5, 29, 67, 173, 174, 277, 282, and 287)

[Bell Laboratories 1979] Bell Laboratories. *Unix Programmer's Manual*, volume 1. 7th edition, 1979. (cited on page 284)

[Berkhin 2002] P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002. (cited on page 64)

[Bertoli et al. 2009] M. Bertoli, G. Casale, and G. Serazzi. JMT: Performance engineering tools for system modeling. *SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009. (cited on pages 61 and 287)

[Bertolino and Mirandola 2004] A. Bertolino and R. Mirandola. CB-SPE tool: Putting component-based performance engineering into practice. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 04)*, volume 3054 of *LNCS*, pages 233–248. Springer, 2004. (cited on pages 2, 67, 282, and 287)

[Bielefeld 2012] T. C. Bielefeld. Online performance anomaly detection for large-scale software systems, 2012. Diploma Thesis, Kiel University. (cited on pages 65, 81, 88, 89, 276, and 300)

[Boskovic and Hasselbring 2009] M. Boskovic and W. Hasselbring. Model-driven performance measurement and assessment with MoDePeMART. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09)*, volume 5795 of *LNCS*, pages 62–76. Springer, 2009. (cited on page 288)

[Box 1998] D. Box. *Essential COM*. Addison-Wesley Professional, 1998. (cited on pages 29 and 282)

[Brambilla et al. 2012] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2012. (cited on pages 3, 11, and 12)

[Briand et al. 2006] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering (TSE)*, 32(9):642–663, 2006. (cited on page 289)

[Brosig et al. 2011] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pages 183–192, 2011. (cited on pages 280 and 289)

[Bruneton et al. 2006] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal component model and its support in Java. *Software: Practice and Experience (SPE)*, 36:1257–1284, 2006. (cited on pages 29 and 282)

[Bunge 2008] S. Bunge. Transparentes Redeployment in komponentenbasierten Softwaresystemen ("Transparent redeployment in component-based software systems", in German), 2008. Diploma Thesis, University of Oldenburg. (cited on pages 84, 87, 89, 246, and 303)

[Bureš et al. 2006] T. Bureš, P. Hnětynka, and F. Plášil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *4th International Conference on Software Engineering Research, Management and Applications (SERA '06)*, pages 40–48. IEEE, 2006. (cited on pages 29 and 282)

[Candea et al. 2004] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation*, 56(1-4):213–248, 2004. (cited on page 290)

[Canfora et al. 2011] G. Canfora, M. Di Penta, and L. Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM (CACM)*, 54:142–151, 2011. (cited on page 288)

[Chen et al. 2002] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '02)*, pages 595–604. IEEE, 2002. (cited on pages 229 and 285)

[Cheng et al. 2009] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. D. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. In B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 1–26. Spinger, 2009. (cited on page 34)

[Cheng 2008] S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2008. (cited on pages 280 and 286)

[Chikofsky and Cross 1990] E. Chikofsky and I. Cross, J.H. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990. (cited on page 288)

[Clark et al. 2007] A. Clark, S. Gilmore, J. Hillston, and M. Tribastone. Stochastic process algebras. In M. Bernardo and J. Hillston, editors, *Proceedings of the 7th International School on Formal Methods for the Design of Computer, Communication, and Software System (SFM '07)*, volume 4486 of *LNCS*, pages 132–179. Springer, 2007. (cited on page 65)

[Clements et al. 2002] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley / Pearson Education, 2002. (cited on pages 22 and 26)

[Community Z Tools Project 2014] Community Z Tools Project. CZT: Community Z Tools — Tools for developing and reasoning about Z specifications. `http://czt.sourceforge.net/`, 2014. (cited on page 92)

[Cortellessa et al. 2008] V. Cortellessa, P. Pierini, R. Spalazzese, and A. Vianale. MOSES: MOdeling Software and platform architecture in UML 2 for Simulation-based performance analysis. In *Proceedings of the 4th International Conference on Quality of Software Architectures (QoSA '08)*, volume 5281 of *LNCS*, pages 86–102. Springer, 2008. (cited on page 287)

[Cortellessa et al. 2011] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-based software performance analysis*. Springer, 2011. (cited on pages 3, 11, 65, 282, and 289)

[Crnković et al. 2011] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering (TSE)*, 37(5):593–615, 2011. (cited on pages 28 and 282)

[Crovella and Bestavros 1997] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking (TON)*, 5(6):835–846, 1997. (cited on page 63)

[Cugola and Margara 2012] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15:1–15:62, 2012. (cited on page 132)

[Czarnecki and Helsen 2006] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3): 621–645, 2006. (cited on page 14)

[Dashofy et al. 2005] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, 2005. (cited on pages 27 and 282)

[de Lemos et al. 2013] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*, pages 1–32. Springer, 2013. (cited on page 34)

[Denning and Buzen 1978] P. J. Denning and J. P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys (CSUR)*, 10(3):225–261, 1978. (cited on page 66)

[Di Marco and Mirandola 2006] A. Di Marco and R. Mirandola. Model transformation in software performance engineering. In *Proceedings of the 2nd International Conference on the Quality of Software Architectures (QoSA 06)*, volume 4214 of *LNCS*, pages 95–110. Springer, 2006. (cited on pages 67 and 289)

[Diaconescu 2006] A. Diaconescu. *Automatic Performance Optimisation of Component-Based Enterprise Systems via Redundancy*. PhD thesis, Dublin City University, Ireland, 2006. (cited on pages 280, 286, 290, and 303)

[Diaconescu and Murphy 2005] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 05)*, pages 44–53. ACM, 2005. (cited on pages 280, 286, and 290)

[Diaconescu et al. 2004] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component based software systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, pages 214–221, 2004. (cited on pages 280 and 286)

[Dąbrowski 2012] R. Dąbrowski. On architecture warehouses and software intelligence. In *Proceedings of the 4th International Mega-Conference on Future Generation Information Technology (FGIT 2012)*, volume 7709 of *LNCS*, pages 251–262. Springer, 2012. (cited on page 277)

[Duboc et al. 2007] L. Duboc, D. Rosenblum, and T. Wicks. A framework for characterization and analysis of software system scalability. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, pages 375–384. ACM, 2007. (cited on page 44)

[Eberlein 2011] S. Eberlein. Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance-Monitoring, 2011. Diploma Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany. (cited on page 277)

[Eclipse Foundation 2014] Eclipse Foundation. Eclipse Modeling Project. `http://www.eclipse.org/modeling/`, 2014. (cited on page 16)

[Efftinge et al. 2012] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: Implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE '12)*, pages 112–121. ACM, 2012. (cited on page 17)

[Ehmke 2013] N. C. Ehmke. Everything in sight: Kieker's WebGUI in action (tutorial). In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDAYS '13)*, volume 1083 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013. (cited on page 300)

[Esper Team and EsperTech, Inc. 2014] Esper Team and EsperTech, Inc. Esper 5.0.0 reference documentation. `http://esper.codehaus.org/esper/documentation`, 2014. (cited on pages 133, 138, and 152)

[Eucalyptus Systems, Inc. 2014] Eucalyptus Systems, Inc. Eucalyptus. http://www.eucalyptus.com/, 2014. (cited on pages 2, 32, and 33)

[Feiler et al. 2003] P. H. Feiler, B. Lewis, and S. Vestal. The SAE Avionics Architecture Description Language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems (MDES '03)*, 2003. (cited on pages 27 and 282)

[Field and Hole 2012] A. Field and G. J. Hole. *How to Design and Report Experiments*. Sage Publications Ltd., 2012. (cited on page 84)

[Fielding 2000] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, CA, USA, 2000. (cited on pages 25 and 26)

[Fielding and Taylor 2002] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002. (cited on page 26)

[Fittkau 2011] F. Fittkau. Infrastructure-as-a-Service (IaaS) with Amazon EC2/Eucalyptus. In *Seminar Software Performance Engineering WiSe 2010/2011, Department of Computer Science, Kiel University, Germany*, 2011. (cited on pages 88, 225, and 240)

[Fittkau 2012] F. Fittkau. Simulating cloud deployment options for software migration support, 2012. Master's Thesis, Kiel University. (cited on pages 88 and 246)

[Fittkau et al. 2013] F. Fittkau, J. Waller, P. Brauer, and W. Hasselbring. Scalable and live trace processing with kieker utilizing cloud computing. In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDAYS '13)*, volume 1083 of *CEUR Workshop Proceedings*, pages 89–98. CEUR-WS.org, 2013. (cited on page 223)

[Fittkau et al. 2014] F. Fittkau, A. van Hoorn, and W. Hasselbring. Towards a dependability control center for large software landscapes. In *Proceedings of the 10th European Dependable Computing Conference (EDCC '14)*, pages 58–61, 2014. (cited on page 301)

[Flaig et al. 2013] A. Flaig, D. Hertl, and F. Krüger. Evaluation of java profiler tools, 2013. Special Research Software Engineering (Fachstudie), University of Stuttgart, Institute of Software Technology, Stuttgart, Germany. (cited on page 285)

[Focke 2006] T. Focke. Performance Monitoring von Middleware-basierten Applikationen. Diplomarbeit, University Oldenburg, 2006. (cited on pages 80, 260, 261, and 276)

[Fowler 2002] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, Reading, Massachusetts, 2002. (cited on page 30)

[Franks et al. 2009] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering (TSE)*, 35(2):148–161, 2009. (cited on pages 61 and 65)

[Frey et al. 2011] S. Frey, A. van Hoorn, R. Jung, W. Hasselbring, and B. Kiel. MAMBA: A measurement architecture for model-based analysis. Technical Report TR-1112, Department of Computer Science, University of Kiel, Germany, 2011. (cited on pages 49, 50, 52, 53, 89, and 283)

[Frey et al. 2012] S. Frey, A. van Hoorn, R. Jung, B. Kiel, and W. Hasselbring. MAMBA: Model-based analysis utilizing OMG's SMM. In *Proceedings of the 14th Workshop Software-Reengineering (WSR '12)*, pages 37–38, 2012. (cited on pages 49 and 89)

[Frotscher 2013] T. Frotscher. Architecture-based multivariate anomaly detection for software systems, 2013. Master's Thesis, Kiel University. (cited on pages 81, 88, 276, and 300)

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (cited on page 170)

[Garlan and Schmerl 2004] D. Garlan and B. Schmerl. Using architectural models at runtime: Research challenges. In *Proceedings of the 1st European Workshop on Software Architecture (EWSA '04)*, volume 3047 of *LNCS*. Springer, 2004. (cited on page 280)

[Garlan et al. 1997] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '97)*, page 7. IBM Press, 1997. (cited on pages 27 and 282)

[Garlan et al. 2003] D. Garlan, S.-W. Cheng, and B. R. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 61–89. Springer, 2003. (cited on pages 35, 280, and 286)

[Garlan et al. 2004] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004. (cited on pages 280 and 286)

[Gmach et al. 2007] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 2007 IEEE International Symposium on Workload Characterization (IISWC '07)*, pages 171–180, 2007. (cited on page 1)

[Gomez-Martinez and Merseguer 2005] E. Gomez-Martinez and J. Merseguer. A software performance engineering tool based on the UML-SPT. In *Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems (QEST '05)*, page 247. IEEE, 2005. (cited on page 287)

[Gorlick and Razouk 1991] M. M. Gorlick and R. R. Razouk. Using Weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE '91)*, pages 23–34. IEEE, 1991. (cited on pages 27 and 282)

[Gorton et al. 2008] I. Gorton, Y. Liu, and N. Trivedi. An extensible and lightweight architecture for adaptive server applications. *Wiley Software: Practice and Experience*, 38(8):853–883, 2008. (cited on page 286)

[Goševa-Popstojanova et al. 2006] K. Goševa-Popstojanova, A. D. Singh, S. Mazimdar, and F. Li. Empirical characterization of session-based workload and reliability for web servers. *Springer Empirical Software Engineering*, 11(1):71–117, 2006. (cited on pages 1, 62, and 63)

[Graham et al. 2004] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 39(4): 49–57, 2004. (cited on page 284)

[Grassi et al. 2007] V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Elsevier Journal of Systems and Software (JSS)*, 80(4):528–558, 2007. (cited on pages 67 and 282)

[Gualtieri et al. 2009] M. Gualtieri, J. R. Rymer, R. Heffner, and W. Yu. The forrester wave™: Complex event processing (CEP) platforms. Technical report, Forrester Research, 2009. (cited on page 138)

[Gul et al. 2008] I. A. Gul, N. Sommer, M. Rohr, A. van Hoorn, and W. Hasselbring. Evaluation of control flow traces in software applications for intrusion detection. In *Proceedings of the 12th IEEE International Multitopic Conference (IEEE INMIC 2008)*, pages 373–378. IEEE, 2008. (cited on page 229)

[Günther 2011] N. Günther. Modellbasierte Laufzeit-Performance-Vorhersage für komponentenbasierte Softwarearchitekturen ("Model-based online performance prediction for component-based software architectures", in German), 2011. Diploma Thesis, Kiel University. (cited on pages 19, 88, 174, 177, 185, and 303)

[Heger 2012] C. Heger. Automatische Problemdiagnose in Performance-Unit-Tests, 2012. Master's Thesis, Karlsruhe Institute of Technology. (cited on page 277)

[Heineman and Councill 2001] G. T. Heineman and W. T. Councill, editors. *Component-based software engineering: Putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., 2001. (cited on page 28)

[Herbst 2012] N. R. Herbst. Workload classification and forecasting, 2012. Diploma Thesis, Karlsruhe Institute of Technology. (cited on pages 89 and 277)

[Herbst et al. 2013a] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *Proceedings of the 4th ACM/SPEC International*

Conference on Performance Engineering (ICPE '13), pages 187–198. ACM, 2013a. (cited on pages 64, 89, and 300)

[Herbst et al. 2013b] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC '13)*. USENIX, 2013b. (cited on page 45)

[Hofmeister 1993] C. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, University of Maryland, 1993. (cited on page 35)

[Hrischuk et al. 1999] C. E. Hrischuk, C. M. Woodside, J. A. Rolia, and R. Iversen. Trace-based load characterization for generating performance software models. *IEEE Transactions on Software Engineering (TSE)*, 25(1): 122–135, 1999. (cited on page 289)

[Huang et al. 1995] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS '95)*, pages 381–390. IEEE, 1995. (cited on page 290)

[Huber et al. 2012] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. S/T/A: Meta-modeling run-time adaptation in component-based system architectures. In *Proceedings of the 9th IEEE International Conference on e-Business Engineering (ICEBE 2012)*, pages 70–77. IEEE, 2012. (cited on pages 89 and 225)

[Huber et al. 2014] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *Springer Service Oriented Computing and Applications (SOCA)*, 8(1):73–89, 2014. (cited on pages 36, 89, 161, 167, 225, 280, and 283)

[Huebscher and McCann 2008] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):1–28, 2008. (cited on page 34)

[IBM 2003] IBM. Web Service Level Agreement (WSLA) language specification, version 1.0, revision: wsla-2003/01/28. `http://www.research.ibm.com/wsla/`, 2003. (cited on pages 1, 46, and 283)

[IEEE 2000] IEEE. IEEE recommended practice for architectural description of software-intensive systems—std. 1471-2000, 2000. (cited on page 22)

[ISO/IEC 2001] ISO/IEC. ISO/IEC 9126: Software engineering – product quality – part 1: Quality model, 2001. (cited on page 40)

[ISO/IEC 2003a] ISO/IEC. ISO/IEC 9126: Software engineering – product quality – part 2: External metrics, 2003a. (cited on page 40)

[ISO/IEC 2003b] ISO/IEC. ISO/IEC 9126: Software engineering – product quality – part 3: Internal metrics, 2003b. (cited on page 40)

[ISO/IEC 2004] ISO/IEC. ISO/IEC 9126: Software engineering – product quality – part 4: Quality in use metrics, 2004. (cited on page 40)

[ISO/IEC 2005a] ISO/IEC. ISO/IEC 20000-1: Information technology – service management – part 1: Specification, 2005a. (cited on pages 1 and 59)

[ISO/IEC 2005b] ISO/IEC. ISO/IEC 20000-1: Information technology – service management – part 2: Code of practice, 2005b. (cited on page 59)

[ISO/IEC/IEEE 2011] ISO/IEC/IEEE. ISO/IEC/IEEE 42010:2011(E): Systems and software engineering — Architecture description, international standard, 2011. (cited on pages 21, 22, 23, and 24)

[Israr et al. 2007] T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Elsevier Journal of Systems and Software (JSS)*, 80(4):474–492, 2007. (cited on page 289)

[Jain 1991] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991. (cited on pages 1, 39, 41, 42, 43, 55, 56, 57, 61, 65, 246, and 287)

[Johnson 1998] M. W. Johnson. Monitoring and diagnosing application response time with ARM. In *Proceedings of the IEEE 3rd International Workshop on Systems Management (SMW '98)*, pages 4–13. IEEE, 1998. (cited on page 284)

[Joint Committee for Guides in Metrology (JCGM) 2008] Joint Committee for Guides in Metrology (JCGM). International vocabulary of metrology — Basic and general concepts and associated terms (VIM), JCGM 200:2008. http://www.iso.org/sites/JCGM/VIM-JCGM200.htm, 2008. (cited on page 55)

[Jouault and Bézivin 2006] F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS '06)*, volume 4037 of *LNCS*. Springer, 2006. (cited on page 19)

[Jouault et al. 2008] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Elsevier Science of Computer Programming*, 72 (1-2):31–39, 2008. (cited on page 19)

[Juse et al. 2003] K. S. Juse, S. Kounev, and A. P. Buchmann. PetStore-WS: Measuring the performance implications of web services. In *Proceedings of the 29th International Computer Measurement Group Conference (CMG '03)*, pages 113–123. Computer Measurement Group, 2003. (cited on page 229)

[Kearney et al. 2010] K. Kearney, F. Torelli, and C. Kotsokalis. SLA*: An abstract syntax for service level agreements. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing (GRID '10)*, pages 217–224. IEEE, 2010. (cited on pages 47, 48, and 283)

[Keller and Ludwig 2003] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003. (cited on pages 46 and 283)

[Kephart and Chess 2003] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003. (cited on pages 33, 34, and 286)

[Kephart and Walsh 2004] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '04)*, pages 3–12. IEEE, 2004. (cited on page 34)

[Kiczales et al. 1996] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming.

Position paper from the Xerox PARC Aspect-Oriented Programming project, Xerox Paolo Alto Research Center, 1996. (cited on page 57)

[Kiczales et al. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 2001 European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *LNCS*, pages 327–354. Springer, 2001. (cited on pages 57, 146, and 284)

[Kieker Project 2014a] Kieker Project. Kieker 1.9 user guide. `http://kieker-monitoring.net/documentation/`, 2014a. (cited on pages 107, 118, 119, 120, 146, 210, 229, and 249)

[Kieker Project 2014b] Kieker Project. Kieker web site. `http://kieker-monitoring.net/`, 2014b. (cited on pages 107 and 119)

[Kiel 2013] B. Kiel. Investigating the use of graph databases for large model repositories, 2013. Master's Thesis, Kiel University. (cited on pages 13, 81, 89, and 130)

[Knoche et al. 2012] H. Knoche, A. van Hoorn, W. Goerigk, and W. Hasselbring. Automated source-level instrumentation for dynamic dependency analysis of COBOL systems. In *Proceedings of the 14th Workshop Software-Reengineering (WSR '12)*, pages 33–34, 2012. (cited on pages 113, 124, and 277)

[Kounev et al. 2010] S. Kounev, F. Brosig, N. Huber, and R. Reussner. Towards self-aware performance and resource management in modern service-oriented systems. In *Proceedings of the 7th IEEE International Conference on Services Computing (SCC '10)*, pages 621–624. IEEE, 2010. (cited on pages 59 and 280)

[Kounev et al. 2012] S. Kounev, S. Spinner, and P. Meier. Introduction to Queueing Petri Nets: Modeling formalism, tool support and case studies (tutorial paper). In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*, pages 9–18. ACM, 2012. (cited on pages 65 and 287)

[Kowall and Cappelli 2013] J. Kowall and W. Cappelli. Gartner's magic quadrant for application performance monitoring, 2013. (cited on pages 58 and 285)

[Koziolek 2010] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Elsevier Performance Evaluation*, 67(8):634–658, 2010. (cited on pages 2, 65, and 282)

[Koziolek and Reussner 2008] H. Koziolek and R. Reussner. A model transformation from the Palladio Component Model to Layered Queueing Networks. In *Proceedings of the SPEC International Performance Evaluation Workshop 2008 (SIPEW '08)*, volume 5119 of *LNCS*, pages 58–78. Springer, 2008. (cited on pages 289 and 303)

[Kramer and Magee 1985] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering (TSE)*, 11(4):424–436, 1985. (cited on pages 36, 280, and 290)

[Kramer and Magee 1990] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering (TSE)*, 16(11):1293–1306, 1990. (cited on pages 36, 280, and 290)

[Kramer and Magee 2007] J. Kramer and J. Magee. Self-managed systems: An architectural challenge. In *2007 Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE, 2007. (cited on pages 34 and 280)

[Krogmann 2010] K. Krogmann. *Reconstruction of software component architectures and behaviour models using static and dynamic analysis*, volume 4. KIT Scientific Publishing, 2010. (cited on pages 289 and 302)

[Kruchten et al. 2006] P. Kruchten, H. Obbink, and J. Stafford. The past, present, and future for software architecture. *IEEE Software*, 23(2):22–30, 2006. (cited on page 21)

[Kuhn et al. 2013] T. Kuhn, H. V. Le, P. Scheide, P. Strobel, C. Waldvogel, K. Wenz, and N. Wolter. KARMA: Kieker Analysis Repository Metamodel Application, 2013. Master's development project. University of Stuttgart, Institute of Software Technology, Germany. (cited on page 302)

[Kühne 2006] T. Kühne. Matters of (meta-) modeling. *Springer Software & Systems Modeling (SoSyM)*, 5(4):369–385, 2006. (cited on page 13)

[Liggesmeyer 2002] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002. (cited on page 40)

[Lilja 2005] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005. (cited on pages 39, 41, 55, 56, and 57)

[Luckham and Vera 1995] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering (TSE)*, 21(9):717–734, 1995. (cited on pages 27 and 282)

[Ludewig 2003] J. Ludewig. Models in software engineering—An introduction. *Springer Software and Systems Modeling (SoSyM)*, 2(1):5–14, 2003. (cited on page 11)

[Ludewig and Lichter 2010] J. Ludewig and H. Lichter. *Software Engineering — Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2nd edition, 2010. (cited on page 11)

[Luk et al. 2005] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200. ACM, 2005. (cited on page 284)

[Magedanz 2011] F. Magedanz. Dynamic analysis of .NET applications for architecture-based model extraction and test generation, 2011. Diploma Thesis, Kiel University. (cited on pages 88, 124, and 277)

[Magee et al. 1995] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC '95)*, pages 137–153. Springer, 1995. (cited on pages 27 and 282)

[Marek et al. 2012] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development (AOSD '12)*, pages 239–250. ACM, 2012. (cited on pages 57 and 284)

[Markovets et al. 2013] V. Markovets, R. Dabrowski, G. Timoszuk, and K. Stencel. Know thy source code. In *Proceedings of the 6th Balkan Conference in Informatics (BCI '13)*, volume 1036, pages 128–131. CEUR-WS.org, 2013. (cited on page 277)

[Marwede et al. 2009] N. S. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR '09)*, pages 47–57. IEEE, 2009. (cited on page 229)

[Matevska 2009] J. Matevska. *Architekturbasierte erreichbarkeitsoptimierte Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit*. PhD thesis, Department of Computer Science, University of Oldenburg, Oldenburg, Germany, 2009. (cited on pages 36, 83, 84, 89, 163, 164, 290, and 303)

[Mcilroy 1969] D. Mcilroy. Mass-produced software components. In *Proceedings of Software Engineering Concepts and Techniques*, pages 138–155. NATO Science Committee, 1969. (cited on page 27)

[Medvidovic and Taylor 2000] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering (TSE)*, 26(1):70–93, 2000. (cited on pages 26, 27, and 282)

[Meier et al. 2011] P. Meier, S. Kounev, and H. Koziolek. Automated transformation of component-based software architecture models to Queueing Petri Nets. In *Proceedings of the 2011 IEEE 19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS 2011)*, pages 339–348. IEEE, 2011. (cited on pages 280, 289, and 303)

[Mell and Grance 2011] P. Mell and T. Grance. The NIST definition of cloud computing. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`, 2011. Special Publication 800-145. (cited on pages 2 and 31)

[Menascé 2002] D. A. Menascé. Load testing, benchmarking, and application performance management for the web. In *Proceedings of the*

2002 International Computer Measurement Group (CMG) Conference, pages 271–282. Computer Measurement Group, 2002. (cited on page 285)

[Menascé and Almeida 2002] D. A. Menascé and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, 2002. (cited on pages 1, 39, 41, 43, 55, 56, 57, 59, 63, 64, 162, and 311)

[Menascé et al. 1999] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *Proceedings of the 1st ACM Conference on Electronic Commerce (EC '99)*, pages 119–128. ACM, 1999. (cited on pages 62, 63, and 64)

[Menascé et al. 2004] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall, 2004. (cited on page 65)

[Menascé et al. 2005] D. A. Menascé, M. N. Bennani, and H. Ruan. On the use of online analytic performance models in self-managing and self-organizing computer systems. In *Self-Star Properties in Complex Information Systems*, volume 3460 of *LNCS*, pages 128–142. Springer, 2005. (cited on page 290)

[Mens and Van Gorp 2006] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Elsevier Electronic Notes in Theoretical Computer Science (ENTCS)*, 152:125–142, 2006. (cited on page 13)

[Microsoft, Inc. 2014] Microsoft, Inc. Windows Azure. `http://www.windowsazure.com`, 2014. (cited on pages 2, 32, and 286)

[Mos 2004] A. Mos. *A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications*. PhD thesis, Dublin City University, Ireland, 2004. (cited on page 281)

[Mos and Murphy 2004] A. Mos and J. Murphy. COMPAS: Adaptive performance monitoring of component-based systems. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, 2004. (cited on pages 281 and 284)

[Musa 1993] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, 1993. (cited on pages 2 and 41)

[Nurmi et al. 2009] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, pages 124–131. IEEE, 2009. (cited on pages 2 and 32)

[Object Management Group, Inc. 2003] Object Management Group, Inc. MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01, 2003. (cited on page 15)

[Object Management Group, Inc. 2004] Object Management Group, Inc. Human-Usable Textual Notation (HUTN), version 1.0. http://www.omg.org/spec/HUTN/1.0/, 2004. (cited on page 16)

[Object Management Group, Inc. 2005] Object Management Group, Inc. UML Profile for Schedulability, Performance, and Time (SPT), version 1.1. http://www.omg.org/spec/SPTP/1.1/, 2005. (cited on pages 16, 61, 66, 67, 117, 282, and 287)

[Object Management Group, Inc. 2006] Object Management Group, Inc. CORBA Component Model Specification, version 4.0. http://www.omg.org/spec/CCM/4.0/, 2006. (cited on pages 28 and 282)

[Object Management Group, Inc. 2008] Object Management Group, Inc. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, version 1.1. http://www.omg.org/spec/QFTP/1.1/, 2008. (cited on page 283)

[Object Management Group, Inc. 2011a] Object Management Group, Inc. Meta Object Facility (MOF), version 2.4.1. http://www.omg.org/spec/MOF/2.4.1/, 2011a. (cited on page 16)

[Object Management Group, Inc. 2011b] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT), version 1.1. http://www.omg.org/spec/QVT/1.1/, 2011b. (cited on page 16)

[Object Management Group, Inc. 2011c] Object Management Group, Inc. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1. http://www.omg.org/spec/MARTE/1.1/, 2011c. (cited on pages 16, 62, 67, 117, and 282)

[Object Management Group, Inc. 2011d] Object Management Group, Inc. Business Process Model and Notation (BPMN), version 2.0.1. http://www.omg.org/spec/BPMN/2.0.1/, 2011d. (cited on page 17)

[Object Management Group, Inc. 2012a] Object Management Group, Inc. Object Constraint Language (OCL), version 2.3.1. http://www.omg.org/spec/OCL/2.3.1/, 2012a. (cited on pages 13 and 16)

[Object Management Group, Inc. 2012b] Object Management Group, Inc. Architecture-Driven Modernization (ADM): Structured Metrics Meta-Model (SMM), version 1.0. http://www.omg.org/spec/SMM/1.0/, 2012b. (cited on pages 3, 16, 48, 49, and 283)

[Object Management Group, Inc. 2013a] Object Management Group, Inc. MOF 2 XMI Mapping (XMI), version 2.4.1. http://www.omg.org/spec/XMI/2.4.1/, 2013a. (cited on page 16)

[Object Management Group, Inc. 2013b] Object Management Group, Inc. Unified Modeling Language (UML), version 2.5. http://www.omg.org/spec/UML/2.5/Beta2/, 2013b. (cited on pages 13, 16, 24, 29, 68, 91, and 117)

[Object Management Group, Inc. 2013c] Object Management Group, Inc. Architecture-Driven Modernization (ADM) Task Force. http://adm.omg.org/, 2013c. (cited on pages 15 and 48)

[Object Management Group, Inc. 2013d] Object Management Group, Inc. Model Driven Architecture (MDA). http://www.omg.org/mda/, 2013d. (cited on page 15)

[Okanović et al. 2013] D. Okanović, A. van Hoorn, Z. Konjović, and M. Vidaković. SLA-driven adaptive monitoring of distributed applications for performance problem localization. *Computer Science and Information Systems (ComSIS)*, 10(10):26–51, 2013. (cited on pages 277 and 302)

[Open Grid Forum 2011] Open Grid Forum. Web Services Agreement Specification (WS-Agreement). http://ogf.org/documents/GFD.192.pdf, 2011. (cited on pages 47 and 283)

[OpenStack Foundation 2014] OpenStack Foundation. OpenStack: The open source cloud operating system. http://www.openstack.org/, 2014. (cited on pages 2 and 32)

[Oracle 2004] Oracle. Java Virtual Machine Profiler Interface (JVMPI). http://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/jvmpi.html, 2004. (cited on page 284)

[Oracle 2011] Oracle. Java Virtual Machine Tool Interface (JVMTI). http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/, 2011. (cited on pages 57 and 284)

[Oracle 2014a] Oracle. Java Servlet Technology. http://www.oracle.com/technetwork/java/index-jsp-135475.html, 2014a. (cited on page 122)

[Oracle 2014b] Oracle. Java Platform, Enterprise Edition (Java EE). http://www.oracle.com/technetwork/java/javaee/, 2014b. (cited on page 30)

[Oracle 2014c] Oracle. Java Platform, Standard Edition (Java SE). http://www.oracle.com/technetwork/java/javase/, 2014c. (cited on page 30)

[Oreizy et al. 1998] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 177–186. IEEE, 1998. (cited on pages 35 and 280)

[Oreizy et al. 1999] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999. (cited on pages 33 and 35)

[Oreizy et al. 2008] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: Framework, approaches, and styles. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*, pages 899–910. ACM, 2008. (cited on pages 35 and 280)

[OSGi Alliance 2012] OSGi Alliance. OSGi Core Release 5. http://www.osgi.org/Specifications/, 2012. (cited on pages 29, 190, and 282)

[Page and Kreutzer 2005] B. Page and W. Kreutzer, editors. *The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 1st edition, 2005. (cited on pages 189, 191, 195, and 287)

[Parsons et al. 2006] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end-to-end runtime path tracing for J2EE systems. *IEE Proceedings – Software*, 153(4):149–161, 2006. (cited on page 285)

[Petriu and Woodside 2007] D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Springer Software and System Modeling (SoSym)*, 6(2):163–184, 2007. (cited on pages 62 and 67)

[Petriu and Shen 2002] D. C. Petriu and H. Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS '02)*, volume 2324 of *LNCS*, pages 159–177. Springer, 2002. (cited on page 289)

[Petriu and Woodside 2002] D. C. Petriu and C. M. Woodside. Software performance models from system scenarios in use case maps. In *Proceedings of the 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS '02)*, volume 2324 of *LNCS*, pages 141–158. Springer, 2002. (cited on page 289)

[Platenius et al. 2012] M. C. Platenius, M. von Detten, and S. Becker. Archimetrix: Improved software architecture recovery in the presence of design deficiencies. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*, pages 255–264, 2012. (cited on page 289)

[R Development Core Team 2014] R Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna: R Foundation for Statistical Computing, 2014. (cited on page 208)

[Rausch et al. 2008] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, 2008. (cited on page 29)

[Reussner et al. 2011] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg. The Palladio Component Model. Technical Report Karlsruhe Reports in Informatics

2011,14, Karlsruhe Institute of Technology, Faculty of Informatics, 2011. (cited on page 67)

[Richter 2012] B. Richter. Dynamische Analyse von COBOL-Systemarchitekturen zum modellbasierten Testen ("Dynamic analysis of cobol system architectures for model-based testing", in German), 2012. Diploma Thesis, Kiel University. (cited on pages 88, 124, and 277)

[Rohr 2014] M. Rohr. *Workload-sensitive Timing Behavior Analysis for Fault Localisation in Software Systems*. PhD thesis, Department of Computer Science, Kiel University, Germany, 2014. To appear. (cited on page 261)

[Rohr et al. 2008] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stöver, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE '08)*, pages 80–85. ACTA Press, 2008. (cited on pages 80, 89, 107, 117, and 261)

[Rohr et al. 2010] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In *1st Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10)*, pages 87–92. ACM, 2010. (cited on pages 229, 232, and 276)

[Rolia and Sevcik 1995] J. Rolia and K. Sevcik. The method of layers. *IEEE Transactions on Software Engineering (TSE)*, 21(8):689–700, 1995. (cited on page 65)

[Salehie and Tahvildari 2009] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–42, 2009. (cited on pages 3, 34, and 286)

[Schulz et al. 2014] E. Schulz, W. Goerigk, W. Hasselbring, A. van Hoorn, and H. Knoche. Model-driven load and performance test engineering in DynaMod. In *Proceedings of the Workshop on Model-based and Model-driven Software Modernization (MMSM '14)*, pages 10–11, 2014. (cited on page 140)

[Shams et al. 2006] M. Shams, D. Krishnamurthy, and B. Far. A model-based approach for testing the performance of web applications. In

Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA '06), pages 54–61. ACM, 2006. (cited on page 229)

[Shaw and Clements 1997] M. Shaw and P. C. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC '97)*, pages 6–13. IEEE, 1997. (cited on page 25)

[Shumway and Stoffer 2006] R. H. Shumway and D. S. Stoffer. *Time Series Analysis and Its Applications – With R Examples*. Springer, 2nd edition, 2006. (cited on page 65)

[Skene 2007] J. Skene. *Language support for service-level agreements for application-service provision*. PhD thesis, University College London, 2007. (cited on pages 48 and 283)

[Skene 2014] J. Skene. The SLAng SLA language—A language for ASP SLAs. `http://uclslang.sourceforge.net/`, 2014. (cited on page 48)

[Skene et al. 2004] J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 179–188. IEEE, 2004. (cited on page 48)

[Skene et al. 2010] J. Skene, F. Raimondi, and W. Emmerich. Service-level agreements for electronic services. *IEEE Transactions on Software Engineering (TSE)*, 36(2):288–304, 2010. (cited on pages 48 and 283)

[Smith and Williams 2002] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002. (cited on pages 41, 42, 44, 61, and 67)

[Smith et al. 2005] C. U. Smith, C. M. Lladó, V. Cortellessa, A. Di Marco, and L. G. Williams. From UML models to software performance results: An SPE process based on XML interchange formats. In *Proceedings of the 5th International Workshop on Software and Performance (WOSP '05)*, pages 87–98. ACM, 2005. (cited on page 67)

[Smith 2000] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000. (cited on pages 91 and 92)

[SPEC Research Group 2013] SPEC Research Group. SPEC Research Group Charter, rev. 2.9. http://research.spec.org/mission-and-charter/, 2013. (cited on pages 277 and 278)

[SPEC Research Group 2014] SPEC Research Group. Repository of peer-reviewed tools for quantitative system evaluation and analysis. http://research.spec.org/projects/tools/, 2014. (cited on page 277)

[Spivey 2001] J. M. Spivey. The Z notation: A reference manual. online, 2001. (cited on page 92)

[SpringSource 2014] SpringSource. Spring. http://www.springsource.org/, 2014. (cited on page 122)

[Srivastava and Eustace 1994] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 196–205. ACM, 1994. (cited on page 284)

[Stachowiak 1973] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. (cited on page 11)

[Stahl and Völter 2006] T. Stahl and M. Völter. *Model-Driven Software Development – Technology, Engineering, Management*. Wiley & Sons, 2006. (cited on pages 3, 11, 12, 143, 144, 223, 277, and 302)

[Steinberg et al. 2009] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2009. (cited on pages 17, 18, and 19)

[Stöver 2009] L. Stöver. Ein Laufzeit-Analyse-Framework zur Unterstützung architekturbasierter, dynamischer Adaption von Software-Systemen ("An online analysis framework supporting architecture-based runtime adaptation of software systems", in German), 2009. Diploma Thesis, University of Oldenburg. (cited on pages 79, 81, 88, and 142)

[Strittmatter et al. 2013] M. Strittmatter, P. Merkle, A. Rentschler, and M. Langhammer. Towards a modular Palladio Component Model. In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*, volume 1083 of *CEUR Workshop Proceedings*, pages 49–58. CEUR-WS.org, 2013. (cited on pages 283 and 300)

[Sun Microsystems 2009] Sun Microsystems. SWaP (Space, Watts and Performance) Metric. http://www.sun.com/servers/coolthreads/swap/, 2009. Last retrieved February 12, 2009. (cited on page 44)

[Szyperski et al. 2002] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002. (cited on page 28)

[Tanenbaum and van Steen 2008] A. S. Tanenbaum and M. van Steen. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2nd edition, 2008. (cited on page 30)

[Taylor et al. 2009] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley & Sons, Inc., 2009. (cited on pages 3, 21, 22, 24, 25, 26, 27, 34, 39, 43, and 282)

[Tel et al. 2013] T. Tel, T. Freiberg, and Z. Ünsür. Evaluation of Java monitoring tools, 2013. Special Research Software Engineering (Fachstudie), University of Stuttgart, Institute of Software Technology, Stuttgart, Germany. (cited on page 285)

[The Apache Foundation 2014] The Apache Foundation. Apache CXF. http://cxf.apache.org/, 2014. (cited on page 123)

[The Eclipse Foundation 2014] The Eclipse Foundation. The AspectJ Project. http://www.eclipse.org/aspectj/, 2014. (cited on page 122)

[The Open Group 2013] The Open Group. Application Response Measurement (ARM). http://www.opengroup.org/tech/management/arm/, 2013. (cited on page 284)

[Tosic 2004] V. Tosic. *Service offerings for XML web services and their management applications*. PhD thesis, Carleton University, Ottawa, Ontario, Canada, 2004. (cited on pages 1, 46, and 283)

[Tosic et al. 2002] V. Tosic, K. Patel, and B. Pagurek. WSOL—Web Service Offerings Language. In *Revised Papers from the CAiSE International Workshop on Web Services, E-Business, and the Semantic Web (WES '02)*, volume 2512 of *LNCS*, pages 57–67. Springer, 2002. (cited on pages 46 and 283)

[van Hoorn 2007] A. van Hoorn. Workload-sensitive timing behavior anomaly detection in large software systems, 2007. Master's thesis (Diplomarbeit), Department of Computer Science, University of Oldenburg, Germany. 125 pages. (cited on pages 80, 228, and 261)

[van Hoorn 2009a] A. van Hoorn. Adaptive capacity management for resource-efficient, continuously operating software systems (research abstract). In *Proceedings of the 2009 DFG Research Training Groups Workshop*, pages 30–31, 2009a. (cited on page 77)

[van Hoorn 2009b] A. van Hoorn. Adaptive capacity management for the resource-efficient operation of component-based software systems. In *Proceedings of the 2008 Dependability Metrics Research Workshop, Technical Report TR-2009-002*, pages 7–11. Department of Computer Science, University of Mannheim, Germany, 2009b. (cited on page 77)

[van Hoorn 2014] A. van Hoorn. Supplementary material for dissertation. `http://kieker-monitoring.net/research/projects/slastic/`, 2014. (cited on pages 3, 7, 91, 92, 137, 173, 174, 185, 234, and 297)

[van Hoorn et al. 2008] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In *Proceedings of the SPEC International Performance Evaluation Workshop 2008 (SIPEW '08)*, volume 5119 of *LNCS*, pages 124–143. Springer, 2008. (cited on pages 62, 140, 228, and 230)

[van Hoorn et al. 2009a] A. van Hoorn, W. Hasselbring, and M. Rohr. Engineering and continuously operating self-adaptive software systems: Required design decisions. In *Design for Future 2009: Proceedings of the 1st Workshop of the GI Working Group „Long-Living Software Systems (L2S2)"*, volume 537 of *CEUR Workshop Proceedings*, pages 52–63, 2009a. (cited on pages 77 and 161)

[van Hoorn et al. 2009b] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation

of software systems. In *Proceedings of the 2nd Warm-Up Workshop for ACM/IEEE ICSE 2010 (WUP '09)*, pages 41–44. ACM, 2009b. (cited on pages 77 and 161)

[van Hoorn et al. 2009c] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, Germany, 2009c. (cited on pages 89, 107, 114, 117, 201, 203, 262, 267, and 276)

[van Hoorn et al. 2011a] A. van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss. DynaMod project: Dynamic analysis for model-driven software modernization. In *Joint Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM 2011) and the 5th International Workshop on Software Quality and Maintainability (SQM 2011)*, volume 708 of *CEUR Workshop Proceedings*, pages 12–13, 2011a. (cited on pages 82, 83, and 276)

[van Hoorn et al. 2011b] A. van Hoorn, H. Knoche, W. Goerigk, and W. Hasselbring. Model-driven instrumentation for dynamic analysis of legacy software systems. In *Proceedings of the 13th Workshop Software-Reengineering (WSR '11)*, pages 26–27, 2011b. (cited on pages 83 and 124)

[van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*, pages 247–248. ACM, 2012. (cited on pages 89 and 107)

[van Hoorn et al. 2013] A. van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss. DynaMod: Dynamische Analyse für modellgetriebene Software-Modernisierung. Technical Report TR-1305, Department of Computer Science, Kiel University, Germany, 2013. (cited on page 276)

[van Ommering et al. 2000] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics

software. *IEEE Computer*, 33(3):78–85, 2000. (cited on pages 27, 29, and 282)

[van Solingen and Berghout 1999] R. van Solingen and E. Berghout. *The Goal/Question/Metric Method: A practical guide for quality improvement of software development*. McGraw-Hill, 1999. (cited on page 84)

[Vogel et al. 2013] C. Vogel, H. Koziolek, T. Goldschmidt, and E. Burger. Rapid performance modeling by transforming Use Case Maps to Palladio Component Models. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*, pages 101–112. ACM, 2013. (cited on page 289)

[von Massow 2010] R. von Massow. Performance simulation of runtime reconfigurable software architectures, 2010. Diploma Thesis, University of Oldenburg. (cited on pages 81, 88, 189, 193, 194, 258, and 303)

[von Massow et al. 2011] R. von Massow, A. van Hoorn, and W. Hasselbring. Performance simulation of runtime reconfigurable component-based software architectures. In *Proceedings of the 5th European Conference on Software Architecture (ECSA '11)*, volume 6903 of *LNCS*, pages 43–58. Springer, 2011. (cited on pages 67, 161, 162, 189, 191, 193, 247, and 287)

[Wang et al. 2007] D. Wang, W. Xie, and K. S. Trivedi. Performability analysis of clustered systems with rejuvenation under varying workload. *Performance Evaluation*, 64(3):247–265, 2007. (cited on page 290)

[Weinstock and Goodenough 2006] C. B. Weinstock and J. B. Goodenough. On system scalability. Technical Note CMU/SEI-2006-TN-012, Software Engineering Institute, 2006. (cited on page 44)

[Wert 2012] A. Wert. Uncovering performance antipatterns by systematic experiments, 2012. Master's Thesis, Karlsruhe Institute of Technology. (cited on pages 277 and 301)

[Woodcock and Davies 1996] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science, 1996. (cited on page 92)

[Woodside et al. 2002] M. Woodside, D. Petriu, and K. Siddiqui. Performance-related completions for software specifications. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 22–32. ACM, 2002. (cited on pages 67, 282, and 289)

[Woodside et al. 2007] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering (FOSE '07)*, pages 171–187. IEEE, 2007. (cited on pages 3, 42, and 65)

[World Wide Web Consortium (W3C) 2007a] World Wide Web Consortium (W3C). SOAP, version 1.2. `http://www.w3.org/TR/soap/`, 2007a. (cited on page 31)

[World Wide Web Consortium (W3C) 2007b] World Wide Web Consortium (W3C). Web Services Description Language (WSDL), version 2.0. `http://www.w3.org/TR/wsdl20/`, 2007b. (cited on pages 31 and 46)

[Zheng et al. 2011] Q. Zheng, Z. Ou, L. Liu, and T. Liu. A novel method on software structure evaluation. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Service (ICSESS '11)*, pages 251–254. IEEE, 2011. (cited on page 277)

[Zobel 2012] C. Zobel. Monitoring komplexer verteilter Softwaresysteme, 2012. Master's Thesis, Hochschule Mannheim, University of Applied Sciences. (cited on pages 123 and 277)

[Zuse 1998] H. Zuse. *A Framework for Software Measurement*. Walter de Gruyter, 1998. (cited on pages 40 and 84)