

Self-Adaptive Performance Monitoring for Component-Based Software Systems

Jens Ehlers

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2011

Kiel Computer Science Series (KCSS) 2012/1 v1.0 dated 2012-04-20

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

The author can be contacted via <http://jehlers.de>

Published by the Department of Computer Science at Christian-Albrechts-Universität zu Kiel
Software Engineering Group

Please cite as:

- ▷ Jens Ehlers. *Self-Adaptive Performance Monitoring for Component-Based Software Systems*.
Number 2012/1 in Kiel Computer Science Series. Department of Computer Science,
2012. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

```
@Book{Ehlers12,  
  author = {Jens Ehlers},  
  title = {Self-Adaptive Performance Monitoring  
    for Component-Based Software Systems},  
  publisher = {Department of Computer Science},  
  year = {2012},  
  month = {april},  
  number = {2012-1},  
  isbn = {9783844814477},  
  series = {Kiel Computer Science Series},  
  note = {Dissertation, Faculty of Engineering,  
    Christian-Albrechts-Universit\at zu Kiel}}
```

© 2012 by Jens Ehlers

Herstellung und Verlag: Books on Demand GmbH, 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 the Christian-Albrechts-Universität zu Kiel. 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. Gutachter: Prof. Dr. Wilhelm Hasselbring
Christian-Albrechts-Universität zu Kiel
2. Gutachter: Prof. Dr. Florian Matthes
Technische Universität München

Datum der mündlichen Prüfung: 20. April 2012

Abstract

Effective monitoring of a software system's runtime behavior is necessary to evaluate the compliance of performance objectives. In addition to studying the construction and evolution of software systems, the software engineering discipline needs to emphasize the interest on robust and flexible *software system operation*, including means for *continuous monitoring*.

Though performance is a critical characteristic for software systems, tools addressing *application performance monitoring*, i.e. monitoring the operation of software systems at application level, are rarely used in practice. This prevalent negligence is expressed by the following symptoms: (1) a posteriori failure analysis, i.e. appropriate monitoring data is seldom collected and evaluated systematically before a failure or performance anomaly occurs, (2) inflexible instrumentation, i.e. probes are injected only at a limited number of fixed measuring points such that component recompilation and redeployment are required for future modifications, and (3) inability of tracing in distributed systems, i.e. tracing of user requests or transactions beyond the borders of components or their execution containers is not supported or not applied.

This thesis has emerged in the context of the *Kieker monitoring framework*, which targets the above shortcomings. A finding of our experimental overhead evaluation is that it is feasible to instrument probes at a multitude of possibly relevant measuring points, as long as not all of them are active at the same time during operation. Therefore, this thesis proposes a self-adaptive performance monitoring approach allowing for dynamic activation of probes and measuring points. As typical for autonomic systems, a control feedback cycle manages the adaptation of the monitoring coverage at runtime. The solution is based on OCL-based monitoring rules

that refine the monitoring granularity for those components that show anomalous responsiveness.

The monitoring data includes performance measures such as throughput and response time statistics, the utilization of system resources, as well as the inter- and intra-component control flow. Based on this data, performance anomaly scores for each provided service and component-inherent operation are computed. The presented anomaly scoring algorithms are based on time series analysis and distribution clustering, respectively.

This self-adaptive performance monitoring approach for component-based software systems reduces the business-critical failure diagnosis time, as it saves time-consuming manual debugging activities. The approach and its underlying anomaly scores are extensively evaluated in lab experiments, e.g. using the SPECjEnterprise2010 industry benchmark. The evaluation results, in combination with the implementation of the Kieker tool, demonstrate the feasibility and the practicability of the approach.

Preface

by Prof. Dr. Wilhelm Hasselbring

Traditionally, software engineering primarily addresses the *construction* and *evolution* of software. In addition to studying the construction and evolution of software systems, the software engineering discipline needs to address the operation of continuously running software systems. Means for efficient and robust *operation* are particularly critical for enterprise software systems. Continuous monitoring may enable early detection of quality-of-service problems, such as performance degradation, and may deliver usage data for resource management. Such monitoring information is also required to check the fulfillment of service level agreements. Therefore, a system's runtime behavior should be monitored and analyzed continuously. This can, for instance, be the basis for runtime failure detection and diagnosis via performance anomaly detection.

A requirement for the robust operation of software services are means for effective monitoring and analysis of software runtime behavior. In contrast to profiling for construction activities, monitoring of operational services should only impose a small performance overhead to its runtime performance. Even when a monitoring framework, such as Kieker, only imposes a small overhead per probe, a full instrumentation will not be acceptable for monitoring an operational, industrial-scale system.

In this thesis, a new approach to adaptive monitoring is proposed and extensively evaluated. Based on the observation that the overhead of deactivated probes is negligible, Jens Ehlers invents an adaptive monitoring approach which – under normal conditions – just activates probes at the interfaces of components. Only when performance anomalies are observed

at these coarse-granular services, monitoring is activated for the methods confined within the components (self-adaptive deep dive).

The technical design and the implementation re-uses and integrates many software components and frameworks from various domains and sources. Examples are the Eclipse platform with EMF and OCL, Zest and Graphviz for graph visualization, Markov4JMeter and Faban for workload generation, the R language and environment for statistical computing, and QPME for stochastic modeling and analysis, to name a few. The re-use of such powerful components and frameworks relieves from building the respective functions, but imposes the challenge to check their fitness for purpose and to integrate diverse architectural styles into a coherent whole. The prototype designed and realized in this thesis constitutes a remarkable engineering achievement.

Besides the conceptual and the technical design, this engineering thesis provides an extensive experimental evaluation. This evaluation does not employ some self-crafted exemplar system on which the own approach works well. Instead, two external examples from industry and from an industry consortium are employed. Particularly, the large Java Enterprise benchmark of SPEC (Standard Performance Evaluation Corporation), an industrial standards body for performance benchmarks, deserves mention – which is why this book is a valuable addition to both a researcher’s and an engineers’s library.

*Wilhelm Hasselbring
Kiel, April 2012*

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem and Research Questions	4
1.3	Scientific Contribution	7
1.4	Outline	9
2	Foundations	11
2.1	Component-Based Software Engineering	11
2.1.1	Software Components and Architecture	12
2.1.2	Development Process and Stakeholders	18
2.1.3	Component Models	22
2.2	Software Performance	27
2.2.1	Performance Metrics	28
2.2.2	Performance Drivers	31
2.2.3	Software Performance Engineering (SPE)	32
2.2.4	Software Performance Modeling	39
2.2.5	Software Performance Measurement	53
2.3	Instrumentation	60
2.3.1	Byte Code Instrumentation (BCI)	60
2.3.2	Aspect-Oriented Programming (AOP)	63
2.3.3	Middleware Interception	65
2.4	Self-Adaptive Software Systems	67
2.4.1	Classification Dimensions for Self-Adaptive Systems	69
2.4.2	Control Feedback Cycle	75
3	Self-Adaptive Performance Monitoring Approach	81
3.1	Application Goals of Adaptive Monitoring	82
3.2	Self-Adaptive Monitoring Process and Architecture	85
3.2.1	Probe Injection	85

Contents

3.2.2	Probe Activation	87
3.2.3	Data Collection	89
3.2.4	Data Provision	90
3.2.5	Data Processing	90
3.2.6	Control and Visualization	96
3.2.7	(Self-)Adaptation	96
3.3	Expressing Monitoring Rules	98
3.4	Performance Anomaly Detection	101
3.4.1	Continuous Evaluation of Monitoring Rules	102
3.4.2	Anomaly Detection Challenges	104
3.4.3	Rating of Anomalous Responsiveness	107
4	Continuous Software System Monitoring Integration	125
4.1	Monitoring Agent Design	125
4.2	Analysis Subscription Model	141
4.3	Visualization of Analysis Results	145
4.3.1	Class Dependency	146
4.3.2	Request Traces as UML Sequence Diagrams	149
4.3.3	Analysis of Use Case Navigation Patterns	150
4.3.4	Resource Utilization	152
4.3.5	Online Analytical Processing (OLAP) Integration	152
4.3.6	Performance Anomaly Analysis	153
4.4	On-Demand Monitoring Adaptation	161
5	Experimental Evaluation	167
5.1	Time Series-Based Forecast Model Evaluation	168
5.2	Anomaly Scoring Evaluation	174
5.3	Monitoring and Analysis Cost Evaluation	181
5.3.1	Monitoring Cost	181
5.3.2	Analysis Cost	183
5.4	Integrated Self-Adaptive Monitoring Evaluation	186
6	Related Work	193
6.1	Related Research Approaches	193
6.2	Application Performance Management	197

Contents

7	Conclusions and Future Work	201
7.1	Summary	201
7.2	Lessons Learned	203
7.3	Future Work	205
	References	207

List of Figures

2.1	Composite pattern of component-based systems, cf. [Gamma et al., 1994]	12
2.2	Conceptual model of an architecture description [ISO/IEC/IEEE, 2011]	16
2.3	Roles within the component-based development process . . .	21
2.4	UML meta-classes defining a component, cf. [OMG, 2011e] .	24
2.5	Alternative component notations in UML	24
2.6	Response time decomposition	28
2.7	Scalability curves for response time and throughput	30
2.8	Software component performance drivers, cf. [Becker et al., 2006]	31
2.9	SPE process, based on [Smith and Williams, 2002]	35
2.10	Integration of SPE into the software development process . .	39
2.11	Markov chains	42
2.12	Extended Queuing Network	43
2.13	Queuing Petri Net	45
2.14	Performance analysis process	46
2.15	UML-SPT performance analysis meta-model [OMG, 2005] . .	47
2.16	Sequence and deployment diagram annotated according to UML-SPT	48
2.17	Core Scenario Model (CSM) meta-model [Petriu and Woodside, 2007]	49
2.18	Development process of the Palladio Component Model [Koziolek et al., 2008]	51
2.19	Palladio Component Model RDSEFF example	52
2.20	Software system virtualization layers	57
2.21	Jikes RVM scheduling [Hauswirth et al., 2004]	58
2.22	Comparison of compile-time and load-time weaving	65

List of Figures

2.23	Sensors and actuators of a self-adaptive system, cf. [Rohr et al., 2006]	70
2.24	Adaptation time, types, and techniques [McKinley et al., 2004]	73
2.25	Dependability taxonomy, cf. [Avizienis et al., 2004]	75
2.26	Control feedback cycle of self-adaptive systems	76
3.1	Adaptive monitoring process	86
3.2	Adaptive monitoring architecture, cf. [Ehlers and Hasselbring, 2011b]	87
3.3	Software system with operation intercepting probes	88
3.4	Dynamic call tree with corresponding call graph and calling context tree	91
3.5	Cube and dimensions in a multidimensional data set	94
3.6	Class model of a calling context tree	99
3.7	Succeeding states of monitoring coverage for anomaly detection	103
3.8	Contextual and collective anomalies	105
3.9	Anomaly detection errors: false positives and false negatives	108
3.10	Time series of workload, CPU utilization, and response time mean	109
3.11	Frequency distributions of avg. response times for different sample sizes	117
3.12	Confidence interval for anomaly hypothesis test (Student's t-test)	119
3.13	Local outlier probabilities (LoOP) [Kriegel et al., 2009]	122
3.14	LoOP applied to operation response time distribution	122
4.1	Class diagram of the Kieker Monitoring component realization	127
4.2	Sequence diagram related to the operation interception probe	136
4.3	Class diagram of the Kieker Analysis component realization	142
4.4	Illustration of a request call trace and an operation call	144
4.5	Screenshot of the Kieker Analysis project stream editor	145
4.6	Visualization of effective class dependencies at runtime	147
4.7	Class dependencies extracted by static code analysis	148
4.8	Request trace visualized as a UML sequence diagram	149
4.9	User profile visualization	151

List of Figures

4.10	Resource utilization time series	152
4.11	Online analytical processing (OLAP) integration	153
4.12	Performance anomaly analysis model	155
4.13	Interactive visualization of a calling context tree	159
4.14	Performance time series visualization	160
4.15	Model of the MonitoringAdaptation plugin	161
4.16	Kieker plugin for runtime monitoring adaptation	162
5.1	Forecast model evaluation – JPetStore scenario setup	169
5.2	Forecast model evaluation – SPECjEnterprise2010 scenario setup	170
5.3	Line charts of measured and forecasted response times	172
5.4	QPN: System bottleneck’s scheduling impacts service time predictability	174
5.5	Anomaly score evaluation – workload setup	177
5.6	Anomaly score evaluation – results visualization (Scenario AS ₁)	178
5.7	Anomaly score evaluation – results visualization (Scenario AS ₂)	180
5.8	Evaluation of the monitoring cost [Ehlers et al., 2011]	183
5.9	Evaluation of the analysis cost	184
5.10	Call graph of sample system for anomaly injection	187
5.11	Evaluation of the self-adaptive monitoring process	189
5.12	Self-adaptive monitoring process applied to the JPetStore application	191
6.1	Monitoring levels of [Munawar et al., 2008]	195
6.2	APM market as perceived by Gartner [Cappelli and Kowall, 2011]	199

List of Tables

5.1	Forecast model evaluation – mean forecast errors	171
5.2	Anomaly score evaluation – results table	179
6.1	Commercial application performance monitoring tools	198

Listings

2.1	Corresponding code fragment to Figure 2.19	52
2.2	Load-time byte code instrumentation by attaching a Java agent	61
2.3	Byte code modification example using the Javassist API . . .	62
2.4	AspectJ annotation style aspect definition	64
4.1	Component-declaring annotations	130
4.2	AspectJ pointcut configuration example	138
4.3	Kieker Analysis filter notification pattern	143
5.1	Anomaly score validation – true/false positives/negatives . .	176

Introduction

The following introduction starts with a motivation why self-adaptability of a software system's monitoring coverage at runtime is a demanded capability (Section 1.1). The challenges coming along with the need for self-adaptive monitoring are discussed and used to derive research questions that are studied in the course of this thesis (Section 1.2). Afterwards, the scientific contributions of this thesis are summarized (Section 1.3) and its structure is outlined (Section 1.4).

1.1 Motivation

Enterprise software systems that provide a multitude of transactional and stateful services are characterized by their complexity. Typically, such systems are based on a multi-layered and component-based architecture. In business contexts, many systems are engineered using established component-based middleware frameworks like Java EE or .NET. The component-based approach facilitates the development of reliable and flexible software systems by increasing the reusability and exchangeability of proven, possibly third-party, components. The interdependent components are distributed over different runtime containers and assembled to provide a composite service via synchronous or asynchronous communication protocols. The systems have to serve up to thousands of requests/s in parallel. Further, they are exposed to intensity-varying

1. Introduction

workload that is difficult to forecast and challenges the system's design concerning performance. Performance requirements claim dependable responsiveness and throughput numbers regardless of workload peaks. Thus, analyzing a system's performance at design time and profiling in advance of operation at construction time is not sufficient from an operational perspective. Effective monitoring of a continuously running software system is inevitable for its dependable operation.

For a system user, non-functional requirements like performance and the related availability are critical quality of service (QoS) characteristics. Thus, runtime performance is known to be crucial for the user acceptance and economic success of a software system. Primarily, performance addresses the responsiveness of a software system, and secondary, its throughput and resource capacity utilization under increasing workload (scalability). The users judge a service by the individually perceived responsiveness and are rarely interested in justifications because of high throughput or utilization rates. Accordingly, enterprise software systems are exposed to a performance risk that can be limited by a systematic performance engineering approach.

A client might depend on certain uptime rates, maximum response times, or throughput numbers. Service providers have to assure by contract in a service level agreement (SLA) that objectives like the above are complied to a specified degree regardless of the overall system workload. The compliance of SLAs cannot entirely be verified at design or test time but has to be monitored while the system actually operates.

Though performance is a critical factor, continuous performance monitoring is often neglected in practice. A recent survey among Java practitioners and experts reveals and quantifies this contradiction [Snatzke, 2009]: Application-level monitoring tools that support performance analytics are either unknown and not employed in software projects by two-thirds of the interviewees. Instead, monitoring probes are instrumented only in reaction to prior performance degradations or even system failures. With this reactive approach, it takes some critical time until new monitoring

1.1. Motivation

data is available for analysis. By consequence, manual and time-consuming debugging activities are started in test environments.

Typical problem areas are memory leaks and garbage collection, thread concurrency, remote service calls, database access, and legacy integration. The underlying resource bottlenecks or locking conflicts of the problems on the surface appear in many cases only after a longer operation time or a larger workload intensity. Therefore, they are occasionally missed during profiling and load test activities, and point out the necessity of continuous monitoring.

Due to the complexity of large-scale enterprise software systems (particularly by the distributed deployment and assembly of components realized with heterogeneous technologies), instrumentation and monitoring of a system's control flow and performance characteristics are a challenging and costly task. Thereby, two constraints have to be considered: (1) Instrumentation should be non-intrusive to the application business logic as far as possible. (2) In contrast to profiling, monitoring during operation time should only impose a small performance overhead.

It is difficult to decide at design time where to place the monitoring probes and which data should be collected. A main issue for dynamic analytics addressing software behavior comprehension is the amount of information which is collected and processed at runtime. Potentially, more detailed monitoring data allows for more detailed analyses. On the other hand, instrumentation, data collection and logging, as well as online data analysis cause measurable overhead. A trade-off between the granularity and the overhead of monitoring and analysis at runtime has to be reached. Even when the monitoring overhead is disregarded, an unfiltered information flood is not efficient and helpful to a performance analyst if the information is not processed and visualized in aggregated views in near real-time.

1. Introduction

1.2 Problem and Research Questions

The major portion of the monitoring overhead is not caused by the instrumentation of probes at numerous measuring points in the control flow, but by the contentual complexity of the probe implementations. This assumption is attested by the experimental evaluations in the course of this thesis (see Section 5.3). The evaluations quantify the overhead impact of how monitoring data is collected, logged, and processed in subsequent analyses. A finding is that it is feasible to instrument probes at a variety of possibly relevant measuring points, as long as not all of them are active at the same time during operation. Hence, it is desirable to introduce an adaptable monitoring approach that allows for dynamic activation and deactivation of probes and measuring points at runtime.

The adaptation of the monitoring coverage can either take places manually, conducted by a performance analyst, or autonomically, conducted by the monitoring system itself. The approach provided by this thesis focusses on self-adaptive, i.e. autonomic, monitoring and its scientific and practical challenges. Means for manual monitoring adaptation at runtime are a valueable by-product of the implementation.

A main goal of software system monitoring is the early detection of QoS problems (see Section 3.1), particularly with regard to fault tolerance. The proposed self-adaptive monitoring approach targets fast failure recovery, or at best failure prevention, based on performance anomaly localization. It is well-known that a major part of the failure recovery time is required to locate the root cause of a failure [Kiciman and Fox, 2005]. With manual adaptation of the monitoring coverage, the human decision to change the set of active probes or measuring points is based on a prior interpretation of performance measures, which are typically visualized in a monitoring control panel. A typical situation is that performance time series indicate a decline of a service's throughput though the workload did not increase. A performance analyst who recognizes this incidence is interested in the cause of the phenomenon and therefore tries to activate more measuring

1.2. Problem and Research Questions

points in the affected components. It takes some time until enough monitoring data is collected by the newly activated measuring points. The proposed self-adaptive, rule-based monitoring approach aims at reducing this potentially business-critical wait time that delays the diagnosis of a failure or an internal anomaly. In the best case, an anomaly is localized and reported so early that it can be removed before a failure becomes perceivable to the system users.

The self-adaptive monitoring approach allows zooming into a component on demand if it behaves anomalous. In this case, zooming means to activate more (or less) measuring points in the control flow aiming at increasing (or decreasing) insight, e.g. into the operation call stack, effective loop iterations, or conditional branches taken. For self-adaptive control of the monitoring coverage, an inference mechanism continuously evaluates a set of previously specified monitoring rules (see Section 3.3). The monitoring rules should reflect previously specified monitoring goals. For some goals, it is sufficient to monitor responsiveness and utilization at component level, e.g. to fulfill evidence of SLA compliance, to support capacity planning decisions, or to extract usage patterns for interface design or marketing purposes. Concentrating on the goal to localize the causes of performance anomalies, the component-internal control flow is automatically monitored to gain more detailed analytics data in case of anomaly suspicion.

The following research questions have arisen and have been studied while conceiving the targeted self-adaptive monitoring approach for performance anomaly localization. Section 1.4 outlines where these research questions are tackled in the remainder of the thesis.

Software performance modeling

- #1: How can the composition and the performance measures of a monitored enterprise software system be captured in an integrated model?

1. Introduction

Instrumentation

#2: How should the monitored system be efficiently instrumented for continuous and adaptable runtime behavior monitoring?

Self-adaptability

#3: How should the required self-adaptability feature be integrated into the monitoring framework (control feedback cycle)?

#4: How can continuously evaluated monitoring rules implement such a control feedback cycle?

#5: How can these monitoring rules be expressed?

Performance anomaly localization

#6: Which monitoring data is required to reason about software performance anomalies at runtime?

#7: Which are effective approaches to detect such performance anomalies?

#8: How can the monitoring data be aggregated and presented in appropriate ways to a performance analyst?

Evaluation

#9: How much overhead is caused by extensive software system monitoring?

#10: To which degree of accuracy is it possible to forecast response times of software services?

#11: How effective are the proposed anomaly scoring procedures in comparison?

#12: Is a self-adaptive control of the monitoring coverage feasible in practice?

1.3 Scientific Contribution

The main contributions of this thesis to the research area of software performance engineering are:

1. a rule-based, self-adaptive approach for continuous software system monitoring,
2. different software performance anomaly detection approaches,
3. a proof of concept that includes the design and implementation of 1. and 2. as part of the Kieker monitoring framework¹,
4. and an experimental evaluation of 1. and 2.

In the following, these contributions are outlined in more detail.

Self-adaptive monitoring approach: A rule-based, self-adaptive monitoring approach is proposed that allows on-demand changes of the monitoring coverage at runtime. The monitoring rules follow the goals for which monitoring data is required. The approach concentrates on the monitoring goal to localize performance anomalies that change the valid behavior of a software system as perceived by its users. For the specification of the monitoring rules, the Object Constraint Language (OCL) [OMG, 2010] is employed. The monitoring rules refer to performance measures being part of a comprehensive model of the monitored system extracted at runtime. As some model values such as the performance measures frequently change during operation, periodic evaluations of the monitoring rules are necessary. After each evaluation of the rule set, the conclusion might for example be that some measuring points have to be activated to increase insight into the runtime behavior of a distinct component, in particular if it behaves not as expected. The idea of the self-adaptive monitoring process has been presented in [Ehlers and Hasselbring, 2011b].

¹ The Kieker monitoring framework is primarily developed by the Software Engineering Group of the Christian-Albrechts-University of Kiel. Kieker is an open source project. Its web site is located at <http://kieker.sourceforge.net/>.

1. Introduction

Performance anomaly detection approaches: The proposed software performance anomaly detection approaches weight the degree to which the recent responsiveness of a software service/operation is anomalous. Thereby, anomaly scoring is either based on forecast models from the time series analysis domain, or on a distribution clustering algorithm from the statistical classification domain. In each case, a set of historic response time samples is the basis of valuation. The time series-based anomaly scoring approach has been summarized in [Ehlers et al., 2011].

Proof of concept as part of the Kieker monitoring framework: As a proof of concept, the self-adaptive monitoring approach has been integrated as an extension into the Kieker monitoring framework. Kieker is a framework designed for continuous monitoring in production software systems inducing only a very low overhead. It provides the required monitoring capabilities coupled with tools, such as a control panel for the analysis and visualization of monitoring data. The fundamentals of the Kieker framework have been published in [van Hoorn et al., 2009b], its extension for self-adaptability in [Ehlers and Hasselbring, 2011a,b]. The monitoring adaptation implementation is based on meta-models of the Eclipse Modeling Framework (EMF), which allow for evaluation of OCL-based expressions, incorporating the monitoring rules, on object-oriented instance models at runtime. The monitoring rules addressing self-adaptive monitoring control can reference the proposed anomaly scores as part of the system's dynamic performance model.

Experimental evaluation: The thesis includes extensive evaluations of the self-adaptive monitoring approach and the performance anomaly scoring approaches in lab experiments. The SPECjEnterprise2010² industry standard benchmark and the JPetStore³ reference application are employed as monitored systems under test. In this context, our Kieker monitoring

² SPECjEnterprise is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjEnterprise2010 is located at <http://www.spec.org/osg/jEnterprise2010>.

³ <http://code.google.com/p/mybatis/downloads/list?q=jpetstore>

framework has been accepted as a recommended tool of the SPEC Research Group⁴.

1.4 Outline

The remainder of this thesis is structured as follows:

- **Chapter 2** introduces foundations of component-based software engineering (CBSE), software performance, instrumentation, and self-adaptive systems research. The idea of CBSE, the underlying development process, and common component models are introduced in Section 2.1. Section 2.2 brings the performance of software systems into focus and provides a discussion of performance metrics, -drivers, -modeling, and -measurement. Sections 2.1 and 2.2 address the research question (RQ) #1, as described in Section 1.2. Section 2.3 targets RQ #2 as it presents different techniques of instrumentation (byte code instrumentation, aspect-oriented programming, middleware interception) for the injection of monitoring probes into a software system. Afterwards, Section 2.4 discusses the basic concepts and research classifications of self-adaptive software systems. In particular, the idea of a control feedback cycle, being a foundation for answering RQ #3, is explicated.
- **Chapter 3** presents the main approach of this thesis, envisioning a self-adaptive performance monitoring solution for the continuous operation of component-based software systems. Section 3.1 discusses the goals of adaptive monitoring, before Section 3.2 shapes the self-adaptive monitoring process addressing RQ #3 and the required monitoring architecture itself. Aiming at RQ #4 and RQ #5, Section 3.3 describes how the monitoring rules are expressed and evaluated. Finally, Section 3.4 details the underlying performance anomaly detection algorithms, providing a contribution to RQ #7.

⁴ <http://research.spec.org/>

1. Introduction

- **Chapter 4** summarizes the design of the Kieker monitoring framework, particularly its capabilities targeting self-adaptability of the monitoring coverage. Section 4.1 addresses RQ #6 by explaining the functional principle of the *monitoring agents* and how they transfer the monitoring data to a central analysis controller. Section 4.2 summarizes how the monitoring data is processed based on the pipes-and-filters pattern, until it is graphically visualized as described in Section 4.3 and demanded by RQ #8. Section 4.4 presents the Kieker plugin that allows for the specification of the monitoring rules and that embodies the control cycle for self-adaptability.
- **Chapter 5** contains the results of the experimental evaluation. Addressing RQ #10, Section 5.1 evaluates the time-series based forecasts for response times of software services. Section 5.2 compares the different anomaly detection and scoring algorithms proposed in Section 3.4 (RQ #11). Section 5.3 provides measurements of the monitoring and analysis overhead that has to be accepted (RQ #9). Section 5.4 contributes an integrated evaluation of the self-adaptive monitoring approach based on the continuous evaluation of specified monitoring rules (RQ #12).
- **Chapter 6** presents related work of this thesis. While Section 6.1 discusses related research approaches, Section 6.2 makes a comparison to commercial *application performance monitoring* solutions.
- **Chapter 7** concludes the thesis. It gives a summary of the presented approach and its evaluation in Section 7.1. Section 7.2 discusses the lessons learned, assessing benefits as well as limitations of the approach. In the end, Section 7.3 presents the chances for future work.

Foundations

This chapter considers the universe of fundamental terms and concepts being picked up in the following chapters. The goal is to provide brief definitions, explanations and differentiations that are necessary for further comprehension. Section 2.1 introduces the concept of component-based software engineering and presents common component models. Section 2.2 discusses the performance of a software system, particularly modeling and measurement of performance characteristics. Section 2.3 explicates alternative instrumentation techniques to inject monitoring probes into a software system. Finally, Section 2.4 summarizes the basic concepts and scientific classifications of self-adaptive (or autonomic) software systems.

2.1 Component-Based Software Engineering

Component-based software engineering expresses the paradigm of software engineering becoming a sophisticated engineering discipline. The main idea is to hierarchically break down requirements into basic blocks called components, which implement and encapsulate a single aspect of a software system's wide-ranging functionality. This paradigm enhances the reusability of components in different applications and the exchangeability of alternative components. It is an analogy to classical engineering disciplines, where it is common to reuse previously approved components and best practices. The assembly of components to composite services or

2. Foundations

entire systems remains an architectural task implying the composite design pattern (see Figure 2.1).

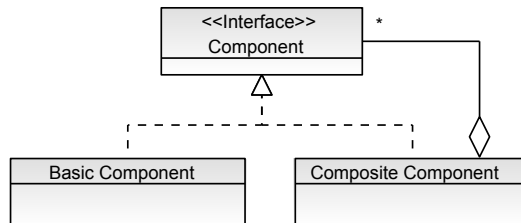


Figure 2.1. Composite pattern of component-based systems, cf. [Gamma et al., 1994]

2.1.1 Software Components and Architecture

In computer science, the term component is widely used in a variety of meanings. To clarify the term in the context of this thesis, two prominent definitions are discussed in the following.

Definition: Software Component by 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.

This definition of Szyperski et al. [2002] is the most frequently cited one, being formulated first in 1996. A newer definition is provided by Taylor et al. [2009].

Definition: Software Component by Taylor et al. [2009]

A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.

2.1. Component-Based Software Engineering

The two definitions do not contradict each other. Even so, they do not state exactly the same characteristics defining a software component. In the following, five essential characteristics found in the definitions are commented, with the last two ones being not demanded by Taylor et al. [2009].

Encapsulated architectural entity: A software component is an entity that encapsulates processing behavior (functionality) and state (structured data). At first glance, this statement does not differentiate components from classes in object-oriented programming. But unlike classes, components are not propagating a concrete implementation paradigm. Components are composed for architectural design purposes. The degree of their inner granularity is not determined. Depending on the designers' needs or perspective, a component can be as simple as a single operation or as complex as a complete system. Instead of mentioning the content of components at all, Szyperski et al. [2002] have chosen an abstract definition as units of composition.

Explicitly defined interfaces: Any component's key facet is that its clients, whether they are human users or other software components, can interact with the component only via an explicitly defined interface. Referring to the design-by-contract principle introduced by Meyer [1997], the interface specifies a contract between the component as a service provider and any service requesting client. If a set of specified preconditions is fulfilled when requesting a service, the server asserts that specified postconditions and invariants are complied. From outside, a client can only see the public interface of a component, while the interior remains a "black box". Consequently, software components embody the software engineering principles of encapsulation, abstraction and modularity.

Explicitly defined dependencies: A software component depends on defined premises concerning its execution context, which are assumed to be permanently complied at runtime. These dependencies might include

- required service interfaces provided by other components in the system,

2. Foundations

- availability of specific resources like data sources or filesystem directories,
- deployment context specifications like a middleware platform, a runtime environment, an operating system (OS), device drivers, and hardware configurations.

Independent deployment: The smallest indivisible entity that is regarded as a basic component is limited by the claim, that a component should be deployable independently of other components.

Third-party composition: Another characteristic of a component is that it is subject to composition by third parties. This is an explicit separation of roles between the provider and the user of a component. Taking of roles is not restricted to organizational boundaries. A provider acts as a developer who implements a component's functionality, specifies its interface and dependencies, and publishes it in a repository, from which clients can retrieve it. Rather than being an end user, a client can act like an architect assembling different components and developing a higher-valued composite service, for which he himself acts as a provider. Reusing distributed components for composition is inspired by the engineering principle to construct complex systems out of detached artifacts.

The process of component assembly and deployment leads to a software system architecture. In 2000, the IEEE Standard 1471-2000 (IEEE 1471) was approved and built a solid terminology for system architectures including the following definition.

Definition: Software Architecture by the IEEE Computer Society [2000]

Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

2.1. Component-Based Software Engineering

Recently, the IEEE 1471 has been replaced by a successor, the norm ISO/IEC/IEEE 42010:2011, which contains further terms being adopted in this thesis:

- Architectural description: Work products used to express an architecture.
- Software-intensive system: A collection of components organized to accomplish a specific set of functions, where software contributes essential influences to the design, construction, deployment, and evolution.
- System stakeholder: A person, team, or organization with interests in a system.
- Architecture view: A work product expressing the system architecture from the perspective of specific concerns.
- Architecture viewpoint: A work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific concerns, i.e. patterns or templates to develop and evaluate views by specific modeling languages and analysis techniques.

Figure 2.2 illustrates how the terms are cross-linked to form the conceptual model of the ISO/IEC/IEEE 42010:2011. Remarkable are the separation of a system architecture and its description, as well as the differentiation of views and viewpoints. The norm does not standardize the development process by recommending distinct methods or modeling languages. Instead, it describes possible uses of architecture descriptions such as specification, communication, or documentation and lists characteristics that enable the uses. These characteristics that make out conformance of an architecture description referring to the norm include: (1) its brief identification and overview, (2) identified system stakeholders and their concerns, (3) definitions of the used architecture viewpoints, (4) an architecture view and associated models for each viewpoint, (5) a documentation of correspondences and known inconsistencies among views, and (6) a rationale for architectural decisions.

2. Foundations

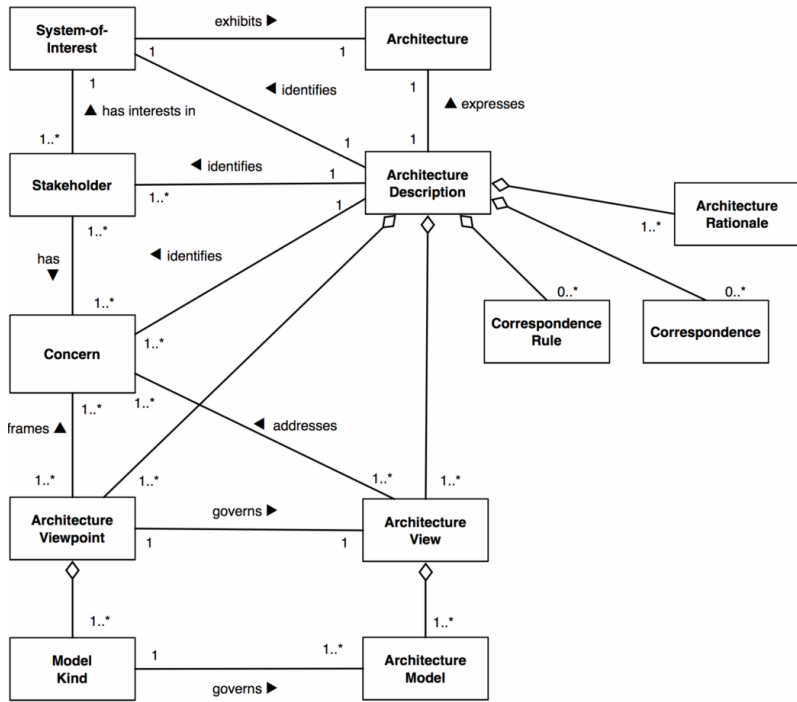


Figure 2.2. Conceptual model of an architecture description [ISO/IEC/IEEE, 2011]

At its essence, a system’s architecture is constituted by a set of principal design decisions made about the system, cf. [Taylor et al., 2009]. During the lifecycle of a system these design decisions evolve. Architecture instances change, fork, and converge. Due to this, a mismatch between a system’s prescriptive and descriptive architecture may arise. The prescriptive architecture is intended by architects at design time, whereas the descriptive architecture stands for its subsequent realization, embodied in a set of components effectively interacting. Major parts of the descriptive architecture can be extracted by runtime monitoring. With our approach, component interactions and dependencies are continuously monitored, in order to control the evolution of a software architecture. This allows the

2.1. Component-Based Software Engineering

detection of architectural erosion caused by implementations that violate the specified prescriptive architecture.

A system's architecture is usually captured in an architectural model making use of a particular notation. Standardized modeling notations are called architecture description languages (ADLs), which differ in characteristic values like textual or graphical, informal or formal, domain specific or general purpose. An architecture model itself is an artifact representing design decisions as a major contribution of a software architect's activity. Several distinct models in different notations may be referring to one system. As architects consolidate interests and decisions of many stakeholders, a model easily gains a high degree of complexity. Especially, if it integrates a variety of concerns at the same time, e.g. interconnections, behavior description, deployment context, and version history of components. This information overload causes a demand for filtering viewpoints, which fade out details and direct attention to individual stakeholder purposes. A viewpoint is defined as a perspective that confines a subset of an architecture related by a common concern, with a view being an instance of a viewpoint for a specific system. The following list provides some example view types:

Logical structural views: Capture the logical entities, like components or classes, the functionality they provide and how they are interconnected, e.g. expressed in UML component or class diagrams.

Behavioral views: Extract parts of the system behavior, including modeling of events, processes, states, and operations, e.g. expressed in UML sequence diagrams or state machines.

Concurrency views: Focus on the management of concurrent threading, synchronization, and scheduling, e.g. expressed in UML activity diagrams or Queueing Petri Nets.

Deployment views: Describe how software components are mapped to execution containers and physical server nodes, e.g. expressed in UML deployment diagrams.

2. Foundations

Physical views: Display the network topology of physical hardware devices.

An early coarse-grained classification of architecture views is the “4+1 Architectural View Model” provided by Kruchten [1995]. Further comparisons and discussions of viewpoint models can be found in Clements et al. [2002] and May [2005].

Consistency among multiple views reflecting on one system has to be assured. An inconsistency occurs when any two views make assertions that cannot be simultaneously true. For instance, these are views on different refinement levels asserting conflicting propositions, or views that describe static and dynamic aspects contradicting each other.

2.1.2 Development Process and Stakeholders

The construction of component-based software systems is part of a complex development process with a variety of stakeholders being engaged. This section provides a brief overview of the activities to be completed before a system is put into operation. This reveals that stakeholders’ interests and expectations of software runtime monitoring depend on their role in the system’s lifecycle.

Inspired by classic process models [Royce, 1987; Boehm, 1988], the Unified Process by Jacobson et al. [1999] has been established as a well-known software development process framework. It is broadly adapted by industry in variants such as the IBM Rational Unified Process (RUP). RUP distinguishes six consecutive engineering disciplines: business modeling, requirements, analysis and design, implementation, test, and deployment. Furthermore, it covers supporting disciplines for project management, configuration and change management, and environment setup. But as several other process models, it misses an explicit operation or maintenance discipline, added in the Enterprise Unified Process by Ambler et al. [2005].

2.1. Component-Based Software Engineering

Unrelated to any concrete process model, the following three trends recently gain increasing acceptance in software development practice [Teiniker et al., 2005]:

- Agile development: Agile methods are appropriate in domains with rapidly changing requirements, so that only short term planning is possible. While a short iteration cycle allows faster responses to change requests, a significant increase of effort for release controlling is implicated. In addition, agility gets lost after some iterations because of risen change cost.
- Model-driven development: Model-driven approaches describe a principle to construct a model of a system, which can later be transformed into finer-grained abstraction levels, particularly into code of a specific programming language. The transformation can be automated by capturing expert knowledge as mapping functions. Today, model-driven development is appropriate for the specification of structures like component interfaces, or class associations and hierarchies. Efficiently constructing nontrivial behavioral models and preserving their consistency is still a challenge being worked on by researchers and tool producers.
- Component-based development: The component-based paradigm is founded on reuse. It is premised on the existence of component repositories providing proven functionality, which has been implemented for previously built systems. A component-based development process has to focus on integrating reusable components rather than constructing functional units from the scratch.

A seminal component-based development process model has been published by Cheesman and Daniels [2000]. It differentiates the following activities:

Requirements: This activity is concerned with surveying business requirements, which are to be facilitated by the software system. A business concept model providing a consistent terminology among users and software engineers is established. Domain experts having profound knowledge of the business domain analyze user demands. A major task is

2. Foundations

the creation of use case models, which describe interactions between the software system and its clients. Domain experts need to quantify expected user behavior as a basis for later workload scenarios testing scalability. Therefore, they are interested in monitoring actual user behavior in related systems or prototype experiments.

Specification: During specification, the component-based software architecture is designed. Software architects categorize and unitize the use cases as inputs from the requirements activity. This decomposition step leads from an overall system specification to detached component specifications and planned component interactions. Thereby, component provisioning constraints and architectural patterns like layer separation have to be considered. If none of the already existing components matches the properties of a designated component, a new component has to be specified and constructed. In this case, the implementation is delegated to developers. An architect's interest in monitoring may be whether or not the prescribed architecture, particularly component-level interactions, are satisfied by the system in operation.

Provisioning: As reuse is the key asset of component-based software development, provisioning takes the place of implementation. Repositories serve as component pools, which are assumed to be easily explorable and deliver a set of frequently used functionalities. In addition, it is expected that the majority of make-or-buy decisions turns out in favor of an existing, possibly marketable third-party component from the repositories. Only components that cannot be acquired have to be implemented by a developer. Developers start construction according to the component specification from the previous activity, which is given as input. They do not clearly know the entire context in that the component will be used. Ideally, it is assumed that developers work for a general-purpose market and are not triggered by a custom order. Therefore, dependencies and restrictions should be kept as small as possible, e.g. concerning available resources, execution container, hardware configuration, or service parametrization. Consequently, developers are interested to monitor how their components behave in certain real-life scenarios regarding responsiveness and scalability.

2.1. Component-Based Software Engineering

Assembly: The components selected in the provisioning activity are assembled conforming to above-mentioned architectural specifications. This may include configuration of containers, customizing of frameworks, or attaching adapters to bridge mismatching interfaces. The output of this activity is a runnable software system.

Test: The assembled software system is tested, reflecting whether or not it fulfills the surveyed requirements. For this reason, all use cases are simulated in a testbed to compare actual and target outcomes. Profiling allows testing of scalability issues, but often a real-life workload is difficult to predict. After exhaustive testing the system is ready to be deployed.

Deployment: Via deployment, the system is placed into its operative environment with real users requesting the provided services. Hence, deployment comprises required installations, environmental configurations, and roll-outs in case of rich client software. Sometimes, even the task of instructing users to make them capable of working with the new system is meant to be included by the term deployment.

The dedicated contributions of involved stakeholders is illustrated in Figure 2.3. Developers provide components into catalog-like repositories. Software architects construct composite services made up of these components. Deployers place these software services into their productive execution environment. Thereupon, users start addressing the service's external interface. Their usage profile is predicted and evaluated by domain experts.

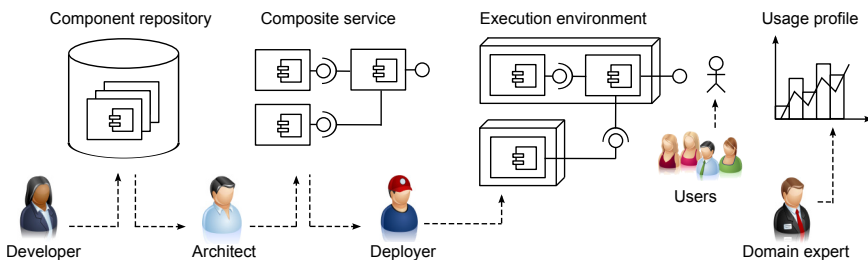


Figure 2.3. Roles within the component-based development process

2. Foundations

2.1.3 Component Models

When component-based development comes to application, a decision to employ one or more specific component model(s) is required. A component model provides a specification that describes how to textually or graphically formulate components and their architectural composition. Thus, the term component model may be misleading, as it addresses a general meta-model for architecture specification and not a concrete system's model. A definition is given by Lau and Wang [2005].

Definition: Software Component Model by Lau and Wang [2005]

A software component model should define (1) the syntax of components, i.e. how they are constructed and represented, (2) the semantics of components, i.e. what components are meant to be, and (3) the composition of components, i.e. how they are composed or assembled.

Existing component models are designed to establish goal-oriented abstractions, which are directed towards specific purposes such as documentation, (model-driven) implementation, transformation, or analysis.

Industrial component models: Industrial component models like Microsoft's COM¹ and .NET², OMG's CORBA Component Model (CCM)³, Enterprise JavaBeans (EJBs)⁴ as well as related approaches like OSGi⁵ and the Spring Framework⁶ concentrate on (platform-dependent) implementation and deployment of components. Implementation standards range from coding templates and interfaces to protocols for remote inter-component communication. To some degree, these meta-models imply a specific runtime infrastructure, e.g. a Java EE application server

¹ <http://www.microsoft.com/com/>

² <http://www.microsoft.com/net/>

³ <http://www.omg.org/spec/CCM/4.0/>

⁴ <http://jcp.org/en/jsr/detail?id=220>

⁵ <http://www.osgi.org/Specifications/>

⁶ <http://www.springsource.com/>

2.1. Component-Based Software Engineering

or an object request broker in CORBA, which provide services such as message communication, transaction management, persistence, or security. As industrial component models are applied in a variety of software engineering projects, a decisive criterion is their practicability assessed by the target audience. By consequence, the meta-models particularly make use of the widespread class concept originated by the object-oriented programming paradigm and enrich it with characteristics of the component definition. This extended reference can for example be found in the use of annotated “plain old Java objects” (POJOs) by EJB and Spring to represent components. The mentioned industrial component models do not allow the definition of different viewpoints, particularly applicable for non-functional analyses. However, they are used as target models for transformations from other component model types described in the following.

Unified Modeling Language (UML): A different scope, originally directed towards documentation and communication of software design, is given by OMG’s Unified Modeling Language (UML). Having emerged from the consolidated notations of Booch [1994], Jacobson [1992], and Rumbaugh et al. [1991], the UML is today’s standard (ISO/IEC 19501) as a language for visualizing, specifying, constructing, and documenting the artifacts of a software system. Essentially, the UML provides a unified notation attached with semantics and a comprehensive meta-model definition. The meta-model has to cope with the difficulties of being supplemented lately to the historically grown notations. In the recent version UML 2.4.1, fourteen diagram types serve as different viewpoints on static and dynamic aspects. Mature tool support has lead to a broad industrial usage.

The UML component model is defined in the UML superstructure specification [OMG, 2011e], parts of which are shown in Figure 2.4. It specifies a component to be a specialized class, which represents an exchangeable entity of a software system. As depicted in Figure 2.4, the UML Component is derived from the UML meta-class Class and contains a set of realizations, with each realization referencing a Classifier. This implies that components are internally realized as classes or composed of other nested subcomponents. Like classes, components are instantiable and encapsulate

2. Foundations

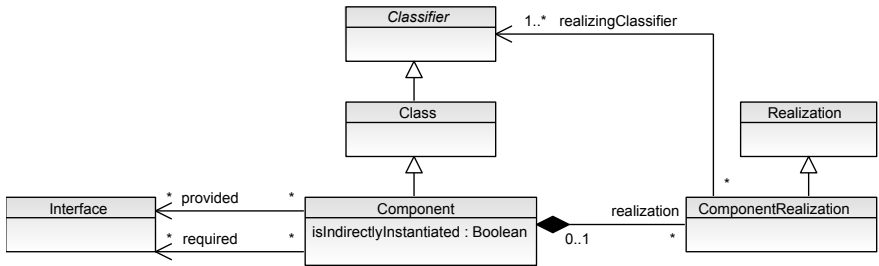


Figure 2.4. UML meta-classes defining a component, cf. [OMG, 2011e]

functional coherent behavior. The explicitly aspired substitutability (at design time and runtime) differentiates a component from a general class. Instead of classes that possess many dependency-causing associations among each other, components are envisioned to be independent from each other. For instance, they should communicate by means of the observer pattern. Besides its realizations, a component is made up of two sets of interfaces. One set contains the provided and the other the required component interfaces. The provided services are categorized by Oesterreich and Bremer [2009] into (1) factory services to create and load component instances, (2) observer services to register event listeners, (3) and object services to provide functional operations. Figure 2.5 illustrates the alternative notations of components in UML.

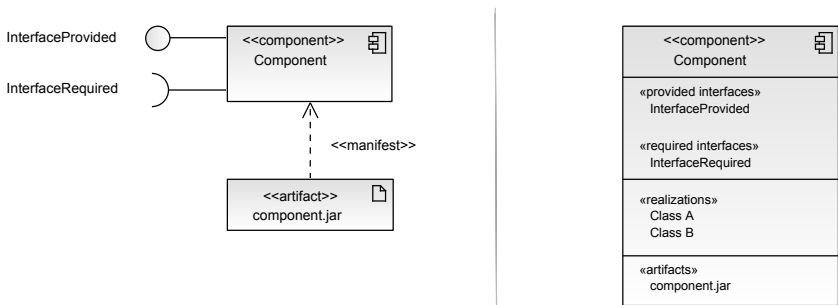


Figure 2.5. Alternative component notations in UML

2.1. Component-Based Software Engineering

The UML meta-model does not support properties for non-functional analyses. To add performance modeling capabilities to the UML, profile extensions such as SPT [OMG, 2005] and MARTE [OMG, 2011f] have been established (see Section 2.2.4).

Architecture Description Languages (ADLs): A lot of preliminary research on how to capture software architectures has been conducted in the 1990s. The result of which was a proliferation of so called ADLs, accompanied by a debate what really makes out a notation or modeling approach to be called ADL. Medvidovic and Taylor [2000] classified and judged ADLs available at the time of their survey. They found that ADLs have in common semantically accurate modeling support for components as well as appropriate connectors, interfaces, and configurations. Early ADLs such as Darwin [Magee et al., 1995], Rapide [Luckham et al., 1995], Wright [Allen and Garlan, 1997], and Weaves [Gorlick and Razouk, 1991] indeed delivered notations precise in semantics, but lacking expressiveness and flexibility. None of these research projects is still active today. Besides, domain-specific ADLs evolved that have been tailored to user needs by giving up genericity in favor of integrating domain know-how into the ADL's semantics. Examples include Koala [van Ommering et al., 2000], which has been developed by Philips to model product lines of consumer electronics, and AADL [Feiler et al., 2003], produced for avionics industry to model embedded real-time systems.

There is a natural trade-off between the expressiveness of a general-purpose modeling language like UML and the demand of semantic precision, which reduces ambiguity and raises analyzability. A solution to face the imperfection of each generic modeling language is to make it extensible to specific semantic customization. Extensions can be categorized to be lightweight, if they only specialize the given meta-model, or heavyweight, if they cause modifications of the meta-model. First ADLs that allowed to expand entities with accessory properties (lightweight) were Acme [Garlan et al., 2000] and ADML [Spencer, 2000]. The XML-based approach xADL [Dashofy et al., 2005] proposes a kind of domain-specific ADL factory (heavyweight). In xADL, users can create an ADL meeting their individual

2. Foundations

needs by selection and adaptation of reference templates, which represent xADL's meta-model in form of a set of XML schemas. Regarding UML, lightweight extensions are possible by means of profiles that allow the definition of analysis-oriented stereotypes, constraints, and tagged values. Example profiles for the industrial component models EJB, .NET, COM, and CCM can be found in the annex of the UML superstructure specification [OMG, 2011e]. Heavyweight extensions would alter the UML meta-model.

A recent, promising software architecture meta-model is the OMG Knowledge Discovery Meta-Model (KDM) [OMG, 2011a]. KDM targets architecture-driven modernization and claims to describe all relevant facets of knowledge related to a software system. KDM provides the following four layers:

- *Program elements layer*, describing the fine-grained code structures determined by the programming language (e.g. data types, procedures, classes, methods, variables) and statements including the control and data flow (comparable to an abstract syntax tree)
- *Resource layer*, describing the applications' runtime platform (e.g. operating system, middleware) and its influence on the control flow, as well as user interfaces, system events, and data storage (e.g. database schemes)
- *Abstractions layer*, representing the structure of a system into subsystems, components, and layers, the system's build and deployment process, as well as relevant business domain knowledge and rules
- *Infrastructure layer*, containing a common core for all other layers and their interconnection, based on the OMG's standard for model-driven engineering Meta Object Facility (MOF) [OMG, 2011d]

The Eclipse MoDisco project⁷ [Bruneliere et al., 2010] supports the extraction of KDM model instances from given (legacy code) artifacts.

Component-based software architecture models have to be distinguished from holistic enterprise architecture modeling approaches, such as

⁷ <http://eclipse.org/MoDisco/>

2.2. Software Performance

ArchiMate [Jonkers et al., 2004; The Open Group, 2009a], TOGAF [The Open Group, 2009b], or the early Zachman framework [Zachman, 1987]. These approaches go beyond software system modeling and provide integration with business process representations, inspired by notations such as BPMN [OMG, 2011b] or Event-driven Process Chains [Scheer, 2000].

Component models particularly directed towards performance modeling such as the Core Scenario Model [Petriu and Woodside, 2007] or the Palladio Component Model [Becker et al., 2009] are discussed in Section 2.2.4.

2.2 Software Performance

Quality is defined as the degree to which a set of inherent characteristics fulfills requirements [ISO, 2005]. Thus, a software system's quality of service (QoS) is determined by its compliance of requirements, in particular user needs. Besides the functional requirements, a software system has to cope with non-functional requirements. While functional requirements determine what the system is supposed to do, non-functional requirements capture how users expect the system to be. That is, non-functional requirements are characteristics that refer to the operation and evolution of a system rather than to specific use cases. Performance is one of the fundamental non-functional requirements. The following definition by Smith and Williams [2002] states how the term performance is interpreted in this thesis.

Definition: Software Performance by Smith and Williams [2002]

Performance is the degree to which a software system or component meets its objectives for timeliness.

Consequently, the basis of any performance metric is time measurement. Timeliness can be partitioned into two dimensions: responsiveness and scalability. According to Smith and Williams [2002], responsiveness is

2. Foundations

a system's ability to meet its objectives for response time or throughput, whereas scalability is the ability to still meet these objectives as the workload increases. The term workload denotes the number and distribution of requests made by the users of a system over time [Jain, 1991].

2.2.1 Performance Metrics

Responsiveness, quantified by the metrics response time and throughput, is typically judged from a user's perspective. Response time describes the amount of time it takes to complete a user request, which is the time between the submission of a request and the receipt of the response. It can be partitioned into network time, queuing time, and service time. The interrelations among the used time terms is shown in Figure 2.6, which consolidates illustrations of Menascé and Almeida [2001] and Jain [1991]. The network time is made up of the time required for the request transmission and latency. The queuing time is the time spent by a request waiting to get access to a resource. The contention for hardware resources, such as CPUs or disk I/O, and software resources, such as thread or database connection pools, is caused by the concurrent arrival and processing of requests. Different scheduling strategies to prioritize queued requests are presented in Section 2.2.4. The service time is the time during which a request is actively served by a resource.

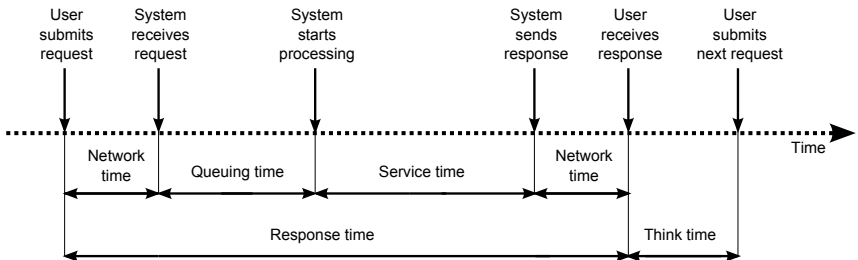


Figure 2.6. Response time decomposition

2.2. Software Performance

Keeping overall response times small is business-critical. Especially in case of human user interfaces, the user's workflow is interrupted while waiting for a response. If response times of frequently used functions are longer than a few seconds, patience is strained and attention can get lost. In the competitive field of web applications or services, users do not accept slow response times and will quickly switch to other service providers. In usability research [Nielsen, 1993], three response time thresholds have been investigated:

- 0.1 second is about the limit below that the user feels a system's reaction to be instantaneous.
- 1 second is about the limit for that the user's flow of thought stays uninterrupted, even though the delay will be noticed.
- 10 seconds is about the limit for that user's attention can be kept focused on a dialog. Longer delays cause the user to perform other tasks in parallel. Thus, feedback is required to indicate until when the system will respond. This is particularly important, if the response time is highly variable, as users want to coordinate their workflows.

An appropriate way to improve responsiveness is displaying interim results or proxy objects. Besides the objectively measured average values, responsiveness is judged subjectively. The individual perception strongly depends on the environmental situation, in which a user interacts with the system. Different session attributes can be used to prioritize requests, especially if SLAs guarantee maximum response times to a group of users.

The throughput of a system is the number of tasks that can be processed in a given time interval. So, throughput is a rate measured in tasks per time unit, e.g. requests/s. As stated, scalability describes the responsiveness while the workload increases. A typical scalability scenario is illustrated by the response time and throughput curves in Figure 2.7. The response time curve shows that below a certain threshold further workload has only a linear effect on the response time. Beyond the threshold, the response time increases exponentially. The reason for this phenomenon is that one or more resources reach 100% utilization and tend to be the bottleneck of the system. Thus, throughput saturates at a maximum value and the system's

2. Foundations

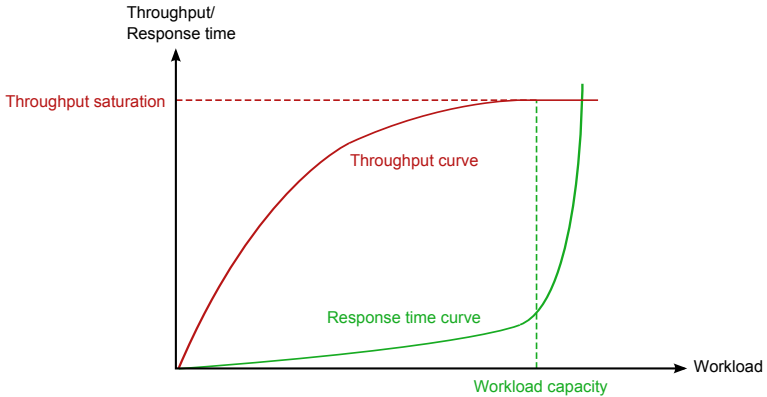


Figure 2.7. Scalability curves for response time and throughput

workload capacity is exhausted. The region where the response time curve changes from a linear to an exponential gradient is called the curve's knee. To provide a scalable service, this knee has to fall behind the workload requirements. Otherwise the bottleneck resources have to be relieved, e.g. through economization that leads to less utilization, or expanded in their capacity, e.g. by means of faster hardware. By consequence, throughput and utilization are metrics that are not only interesting for an entire system, but for each single resource.

The utilization is the rate at which a resource is busy serving requests, i.e. the fraction of busy time in the total time elapsed for measurement. When a resource is not busy, it is idle. It is neither a universal objective to minimize nor to maximize the utilization of a resource. If resources are permanently underutilized, their potential power is not exploited and could be rented or sold to save cost. On the other hand, if resources are already fully utilized while workload further increases, response times will degrade. Monitoring and controlling of response times and throughput on resource level is a prerequisite for a balanced and efficient utilization.

2.2. Software Performance

Regarding web applications or web services, market penetration that leads to more users and more load is usually intended by business decision makers. But if the system capacity does not fulfill the performance requirements, an uptrend in popularity is immediately stopped. An upcoming business model to avoid the required capital expenditure on hardware, software, and related knowhow is associated with the terms Cloud Computing and Software-as-a-Service. The underlying idea is to rent resources tailored to individual demands from external providers, i.e. system capacities increase automatically during load peaks. Pricing is based on periodic fees and actual use (pay per transaction, transferred data volumes, etc.). Nevertheless, the business-critical need of monitoring and controlling the introduced performance metrics is only shifted to the cloud or service provider.

2.2.2 Performance Drivers

This section discusses the factors that influence performance on component-level, inspired by [Becker et al., 2006; Koziolok, 2010]. As outlined in Figure 2.8, the four relevant performance drivers from the perspective of a component (or an entire system) are its realization, deployment, external services, and usage profile.

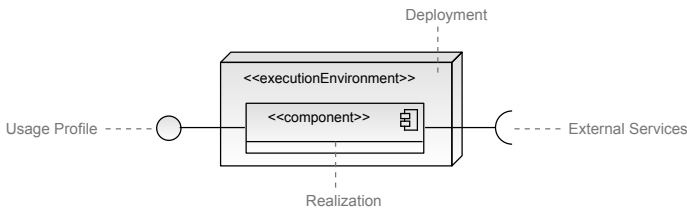


Figure 2.8. Software component performance drivers, cf. [Becker et al., 2006]

Realization: Performance depends on how a component is internally realized. This includes the implemented data structures and algorithms. For instance, the Landau notation is a common approximation of an algorithm's

2. Foundations

limiting behavior and its resulting resource demand of CPU instructions or memory.

Deployment: The execution environment, in which a component is deployed, has a major impact on its performance. This includes the underlying software layers, such as middleware platforms and operating systems, as well as the hardware configuration. Among practitioners, simply allocating more or faster hardware devices is a popular solution approach for performance problems in software systems (known as the “kill it with iron” paradigm). This approach is cost-intensive and effective only if the used algorithms are designed for parallel processing.

External Services: If a component depends on external services, these will influence the performance as it is perceived by the clients of a component. A component developer has to be aware of responsiveness and scalability metrics agreed on with external service providers. In particular, this includes data providers (databases, web services, etc.) towards which queries are directed.

Usage Profile: The performance of a component is affected by the interaction behavior of its clients. Response times and throughput depend on the current workload, i.e. how frequently requests arrive and how much computational cost and data exchange arise from the request parametrization. A component developer has to keep track of a component’s ability to serve multiple requests in parallel. It can be necessary to support priority and scheduling strategies.

2.2.3 Software Performance Engineering (SPE)

The broad range of performance influencing factors listed above documents that performance indeed is a pervasive quality of a software system. The term Software Performance Engineering, describing the discipline to construct software systems that meet their performance objectives, has

2.2. Software Performance

been established by Smith and Williams [2002]. This thesis follows the definition of Woodside et al. [2007].

Definition: Software Performance Engineering by Woodside et al. [2007]

Software performance engineering represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements.

SPE encompasses two research areas providing different approaches to fulfill performance requirements: predictive performance modeling, and performance measurement. Most research projects can be related to one of these two areas. However, research in both areas is likely to converge in future, in order to cover the whole development cycle in a consistent way.

The actual lifecycle stage of a software system determines how to evaluate its performance [Jain, 1991]. Measurements are possible only if at least a prototype or an older version of the system exists. If the system has not been implemented yet, but only conceptualized, modeling is the only choice to evaluate performance at design time. For the audience of a performance evaluation, it is more convincing if the quantitative calibration of a performance model is based on previous measurements, rather than on experiential assumptions only.

The goal of performance studies often is either to compare alternative realizations or to seek for a optimal capacity dimensioning. Performance models can easily be adapted to play through different what-if scenarios. Workload simulations provide insight into the effects and trade-offs of changing single resource properties. Measurements are not that flexible as the real execution environment, particularly physical hardware and network devices, cannot be exchanged as rapidly as in a model. This disadvantage of measurement will become less important since virtualization of hardware resources advances. But certainly, it will remain more time-consuming to adapt, instrument and measure a real system than to run simulations with a system model.

2. Foundations

Among software engineers, the general attitude towards performance statements is very critical. The presumption of innocence is inverted: Until a convincing validation is presented, all performance statements are suspect. Modeling requires simplifying abstractions. Hence, the results of any analytical model evaluation or simulation have to be questioned whether they correspond to the reality. A validation can be done by comparison to measurements or other already accepted performance modeling approaches. Nevertheless, the saleability of results is higher for measurements than for sophisticated performance models. The reason is that the data collection process leading to measured results is easier to understand for outsiders than statistically-founded simulation techniques. This argument is reasonably used as a key justification to employ measurement by means of profiling and monitoring activities. A tabular comparison of pros and cons related to modeling (analytical solving, simulation) and measurements (prototype and final system testing) can be found in [Becker et al., 2009].

Smith and Williams [2002] introduced SPE as a systematic process leading to quantitative evaluations as a basis for capacity procurement. It is aspired that capacities are utilized at a balanced level, as permanent low utilization levels are cost-inefficient and full utilization causes degradation of responsiveness. The SPE approach emphasizes an early consideration of performance requirements, which avoids to fix performance problems in later stages of the software development process. The SPE process includes a sequence of actions, captured in the activity diagram of Figure 2.9:

1. Assess performance risk: Assessing the risk of performance failures supports to decide how much effort planned for SPE is economic. Performance risks that endanger the success of a project can be a cutting-edge application goal, lack of familiarity with relevant technology, inexperienced developers, or a tight schedule. Each performance risk has two factors of impact: the probability of occurrence and the severity of damage in case of occurrence. In general, SPE is less essential if the project is similar to previous ones, if its outcome is not business-critical, and if the demand for hardware resources is minimal.

2.2. Software Performance

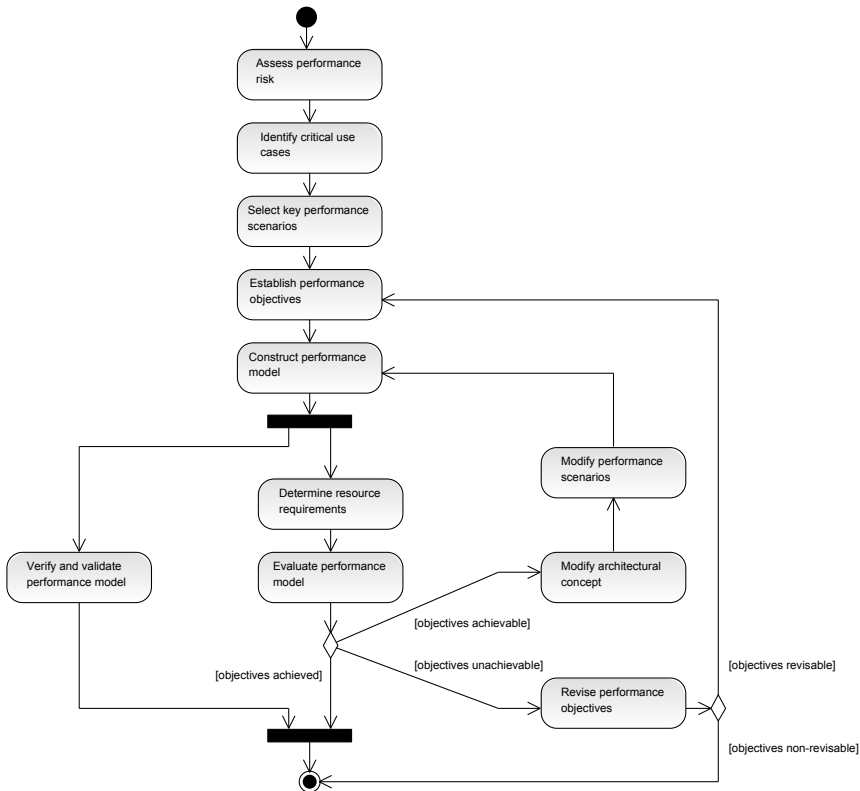


Figure 2.9. SPE process, based on [Smith and Williams, 2002]

2. Identify critical use cases: Not all use cases explored during requirements analysis and specification are connected with a performance risk. The critical use cases include those that significantly impact responsiveness as perceived by the system's users, and those that are important to the continuous operation of the system. Typically, the Pareto principle applies for the usage behavior: A small subset of use cases ($\leq 20\%$) causes the majority of requests and workload ($\geq 80\%$). By consequence, a software system's performance is

2. Foundations

dominated by the frequently called services related to these use cases. Besides, some infrequently used services, which block resources or influence system availability, can have a critical impact on the overall performance risk as well. For example, a recovery service, which is triggered after a system breakdown, has to operate very quickly, although it is rarely used.

3. Select key performance scenarios: For each critical use case, concentrate on the important regular paths, fade out seldom special cases. The selected use case paths have to be interconnected to gain integrated usage scenarios for the whole system. Confidence in the following performance prediction is increased if consensus on the selection of probable, close to reality workload is achieved.
4. Establish performance objectives: Formulating precise performance objectives with reference to workload intensities helps to proof the compliance of performance requirements. Rating of performance risks is possible by comparing objectives to simulation results or measurements. For each usage scenario, measurable threshold values for metrics such as response time, throughput, or utilization are to be defined. By means of different arrival rates for the service calls, the usage scenarios should differ in their workload intensity. For instance, the target response time for a specific service may depend on the number of concurrent users, e.g. max. 0.5 s for 500 users or less, and max. 1 s for up to 1000 users. For a responsible development team, quantified objectives provide an incentive that can evidently be fulfilled and rewarded.
5. Construct performance model: Software performance models enable early detection of inappropriate architectural design. A workable performance model has to be inexpensive in construction and evaluation. Different approaches are discussed in Section 2.2.4. Their general aim is to eliminate the need for a prototype implementation for performance evaluation and to simulate different composition and capacity alternatives.

2.2. Software Performance

6. Determine resource requirements: To predict the performance of a component-based system, the resource demand of each component being involved in the processing of the service calls has to be specified. Resource contention occurs on the level of software resources (e.g. thread or database connection pools, critical section tokens) as well as for hardware resources (e.g. CPU, storage disk I/O, network). The used modeling language has to support both: virtual software resources to represent blocking and synchronization of concurrent processes, and processing hardware resources with inherent scheduling strategies. To quantify resource requirements, the effects of component service calls have to be estimated and specified. If a component's provided service interface is invoked, the resulting resource usage depends on the call's parametrization, which influences the component's internal state, control and data flow. An appropriate modeling language and meta-model is provided with the concept of resource demanding service effect specification (RDSEFF) as part of the Palladio Component Model (see Section 2.2.4). A key problem is that it is difficult to estimate the resource usage of processing steps, being not yet conceptualized in detail during early design stages. In this case, it is practical to work with best- and worst-case estimations and evaluate both estimations in the following action.

7. Evaluate performance model: Performance models are solved analytical or by simulation. If even in the worst-case scenario the performance objectives are achieved (and assumed that the model has been successfully validated meanwhile, see next action), it can be concluded that most likely no performance problems will occur. Otherwise, the analysis results indicate which objectives are possibly not met, and which capacity bottlenecks are the reason for that. The envisioned architecture can be modified. This includes further capacity allocation, as well as alternatives in deployment, component interaction, or realization. Best- and worst-case estimations help to identify components that have a major impact on the system performance if their resource requirements increase. If even the

2. Foundations

best-case estimation does not achieve the objectives and no further iteration of architectural redesign is reasonable, the performance objectives have to be revised. All stakeholders have to acknowledge the new goals. It is also possible to abort a project as it is not feasible to fulfill indispensable performance requirements.

8. Verify and validate performance model: In parallel with construction and evaluation of the model, model verification and validation are ongoing actions. Verification evaluates whether or not the constructed performance model complies with the specified meta-model of the chosen modeling language. Further, it is checked if the model analysis, which includes any transformation, inference, and simulation procedures, is applied correctly. Verification addresses the question [Boehm, 1984]: Is the way how the performance model is build and processed right? On the other hand, validation answers the question: Is the right performance model being build that accomplishes the intended goals? Validation is concerned to assure that the model reflects the performance characteristics of the real system. Thus, it determines how well a model fits for purpose and will be accepted by the stakeholders. Validation requires measurements as a basis of comparison.

In [Smith and Williams, 2003], the actions of the SPE process are augmented and categorized into best practices of project management, performance modeling, performance measurement, and cross-section techniques. SPE has to be integrated into the project schedule and the lived software development process. Figure 2.10 illustrates the integration points of performance modeling and performance measurements into the typical consecutive development activities. Performance modeling takes place during early stages of requirements analysis and design, whereas measurements depend on a first (prototype) implementation. Performance measurements can be separated into the activities of profiling and monitoring (see Section 2.2.5). During profiling and monitoring the effective behavior of a software system at runtime is observed. While continuous monitoring is a requirement for robust operation, profiling

2.2. Software Performance

offers detailed information in the test phase before operation. In contrast to profiling, monitoring of operational services should only impose a small performance overhead.

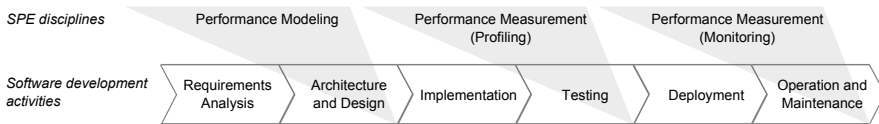


Figure 2.10. Integration of SPE into the software development process

2.2.4 Software Performance Modeling

This section provides a brief overview of analysis-oriented and design-oriented performance modeling approaches. Except for measurements, there are two different methods to evaluate performance models: analytical solutions and simulations [Jain, 1991]. Analytical solutions calculate the performance figures by solving the equation systems of underlying Markov processes. Compared to simulations, the mathematical dissolving process is faster and perfectly reliable, but it necessitates simplifying model assumptions, e.g. exponentially distributed service times and arrival rates. These simplifications lead to predictions that often do not reflect the real system's behavior. Furthermore, analytical model evaluations quickly suffer from state space explosion if the model is nontrivial [Cortellessa et al., 2011]. For simulation purposes, a sample of representative replications of the performance model is instantiated and evaluated. Each simulation run is based on events such as service request arrivals and completions, which are randomly generated according to the probability distributions specified in the model. During discrete-event simulation, the operation of a system is abstracted as a chronological ordered list of events. Each event marks a timestamp when the system state changes. A simulation is not performed in real-time, instead the future event list is sequentially processed and updated. Along the way, the service and queuing times of requests at different resources are aggregated.

2. Foundations

Simulation is much faster than really running the system with generated workload. It is appropriate, if the constraining assumptions of analytical solutions are not acceptable, or if a complete enumeration of all possible system states is prohibitive [Banks et al., 2009].

Analytical solutions and simulations are addressed by analysis-oriented modeling techniques such as Queuing Networks and Petri Nets. The conceptual ideas and restraining conditions of these basic models are briefly outlined in the following. Detailed mathematical formalisms, explicated in [Stewart, 2009] and [Bernardo and Hilston, 2007], are left out for concision. For these basic models, there exists a bunch of analytical solvers or simulation tools to assess performance figures, e.g. QPME⁸ [Kounev and Dutz, 2009], and JMT⁹ [Bertoli et al., 2009].

Markov Chains

Markov chains are the theoretic basis of analytical performance models. A Markov chain is a stochastic process fulfilling the Markov property, which states that a future state depends only on the present state and not on any past states [Stewart, 2009]. Being a stochastic process means that the state transitions are probabilistic. The state space of a Markov chain is discrete. It can be distinguished between Markov chains in discrete and continuous time. In discrete-time Markov chains, state changes take place at deterministic points in time. In contrast, continuous-time Markov chains imply that transitions to the next state can occur at any stochastic time.

Discrete-time Markov chains are represented by a time-independent transition matrix P . For each possible state i , this matrix holds a probability p_{ij} to reach any other state j from i in the next instant of transition. Therefore applies $P = (p_{ij})$ with $\sum_j p_{ij} = 1 \forall i$. The overall system state at time t is expressed by a vector π_t , which quantifies a probability for each state i that the system is currently in this state. If the system processes entities such as service requests, π_t with $\sum_i \pi_t(i) = 1$

⁸ <http://descartes.ipd.kit.edu/projects/qpme/>

⁹ <http://jmt.sourceforge.net/>

2.2. Software Performance

shows how many entities are in which state at the moment of t . After the next state change, the new state probabilities are $\pi_{t+1} = \pi_t P$. For analysis, the system behavior in the long run is of interest. There might exist a stationary state distribution π^* , also called steady state, towards which the process converges if it runs infinitely long. A steady state is determined by satisfying the condition $\pi^* = \pi^* P$ and exists, if all states in the Markov chain are recurrent, i.e. after leaving a state the same state will certainly be reached again sometime. Otherwise, the process includes transient states that are eventually never reached again once they are left. A reason for transience are absorbing states, which cannot be left anymore once they are reached (for example state S_0 in left part of Figure 2.11).

Continuous-time Markov chains are represented by an intensity matrix Q , which holds scheduling transition rates λ_{ij} from each state i to any other state j . On its main diagonal, the intensity matrix contains the negative rates leading away from the respective states, so that $\sum_j \lambda_{ij} = 0 \forall i$. The memorylessness claimed by the Markov property constricts the time periods between the state changes to be exponentially distributed. Thus, each given transition rate parameterizes an exponential distribution, which indicates how much time is expected to elapse till the next state transition. The stationary distribution π^* satisfies $\pi^* Q = \vec{0}$.

Examples of a transition and intensity matrix are shown in Figure 2.11. Besides, the example Markov chains are visualized as directed graphs, in which the vertices represent states and the edges represent transitions labeled with probabilities p or rates λ . The continuous-time Markov chain shows a birth-death process for a queue limited in capacity, with λ_1 being the birth rate and λ_2 being the death rate.

As Markov chains for complex systems quickly become very large, the underlying linear equation systems burst and cannot be computed efficiently anymore. Heuristic solution techniques have been developed to provide approximate solutions. In practice, Markov chains are not modeled manually by performance engineers, since they feature a very low level of abstraction. Instead, other performance modeling techniques such as

2. Foundations

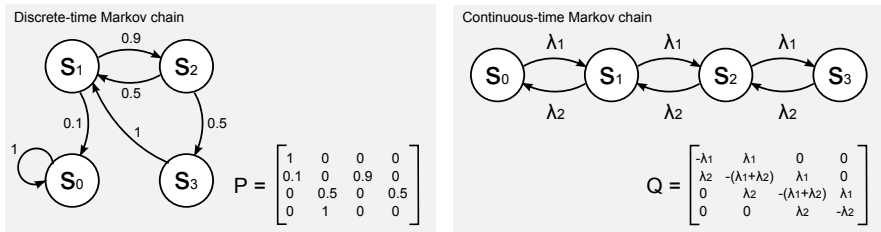


Figure 2.11. Markov chains

Queuing Networks and Petri Nets can be mapped to Markov processes for numerical solution [Bolch et al., 2006; Bernardo and Hilston, 2007]. Markov chains are used to capture a system's effective user profile in Section 4.3.

Queuing Networks (QNs)

Queuing theory has been introduced in the 1970s [Kleinrock, 1976; Buzen, 1971]. At first, Queuing Networks have been applied to model and analyze congestion in communication and production systems, before they were transferred to capacity planning of software systems [Menascé and Almeida, 2001]. As surveyed by Balsamo et al. [2004], most recent performance prediction approaches make use of QN models.

A QN consists of a collection of service centers, which represent processing resources, and probabilistic routes interconnecting the service centers. Custom jobs like user requests travel through the network making use of the provided services, while they seize the respective resources. Each service center is composed of one or more servers, each of which can process one job simultaneously, and a queue, which manages the waiting jobs. Jobs can be partitioned into multiple classes, which might have different service demands and priorities at the service centers. An example QN, depicted in Figure 2.12, shows a topology, where an application server and a database server process incoming user requests. In advance, a request has to allocate a thread from a limited thread pool. Only half of the requests cause database queries. The users' think time between subsequent requests of one session

2.2. Software Performance

is represented by an artificial delay server indicated with a clock symbol. To model the start and end of sessions, the network has an open workload. New jobs can arrive from outside the system and already entered jobs eventually depart. The alternative would be a closed workload, where a specific number of jobs permanently circulates in the network.

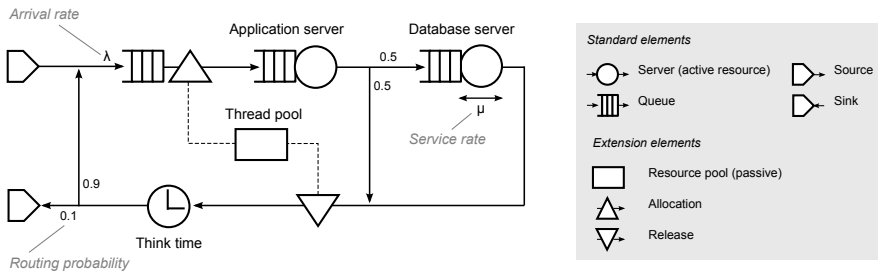


Figure 2.12. Extended Queuing Network

The characteristics of service centers are typically expressed in the form of Kendall's notation $1/2/3/[4]/[5] - [6]$, which enfolds the following parameters, cf. [Fortier and Michel, 2003]:

1. Distribution of interarrival times for service requests, e.g. $M \hat{=}$ Markovian (exponentially distributed), $D \hat{=}$ deterministic, $G \hat{=}$ general. Arrival rates are quantified by a parameter λ .
2. Distribution of service times, e.g. M, D, G . Service rates are quantified by a parameter μ .
3. Number of equal servers in a service center, where m specifies a finite value ≥ 1 .
4. Queuing capacity, which is the maximum number of requests allowed in the service center. Optional, default is ∞ .
5. Number of requests allowed to arrive at the service center, before it stops processing. Optional, default is ∞ .
6. Scheduling discipline. Optional, default is first-come-first-served (FCFS). Alternatives are processor sharing (PS) as an idealized form of round-robin scheduling, preemptive or non-preemptive priority

2. Foundations

scheduling, random scheduling, or infinite server (IS) assuming that no queuing is required.

There exist efficient analytical solutions, such as the convolution algorithm and the mean-value analysis, for product-form QNs. These are networks that satisfy the local balance property, postulating that the arrival rates of each service center are equal to its departure rates. It can be shown that this property is satisfied by the following network types: $M/M/m - FCFS$, $M/G/1 - PS$, and $M/G/\infty - IS$ [Bolch et al., 2006]. However, the local balance property is very restrictive concerning its assumptions (Markovian arrival or service times, infinite queuing capacity, closed workload) and does often not cope with the requirements of real system models. Models with weaker assumptions have to be simulated. The result of a QN analysis provides predictions for the utilization of resources, average queue lengths, and the expected queuing and service times at each service center, which can be summed up to the system's overall response time. QNs have been extended to allow the simulation of multiple resource possession and simultaneous resource consumption [Liehr and Buchenrieder, 2009]. Particularly, the allocation of passive resources as introduced by Sauer et al. [1980] is needed to model token pools for thread synchronization and blocking in software systems (see Figure 2.12).

Extended Petri Nets

Petri Nets are another basic performance modeling technique. Ordinary Petri Nets are appropriate to model synchronization and blocking aspects, but service times and scheduling strategies are originally not supported [Kounev and Buchmann, 2003]. To overcome this shortcoming, several extensions such as Colored, Timed, and Queuing Petri Nets have been established [Jensen and Kristensen, 2009].

A Petri Net is a bipartite directed graph composed of places that contain places P and transitions T . Incidence functions I^- , I^+ specify the interconnections between places and transitions. To complete a Petri Net tuple (P, T, I^-, I^+, M_0) , a start state is defined by an initial marking M_0 . A marking describes the current distribution of tokens representing the

requests, which are processed by the system. Tokens are moved along the interconnections, if transitions fire. Colored Petri Nets (CPNs) introduce different types (colors) of tokens and type-dependent transition firing. Generalized Stochastic Petri Nets (GSPNs) add the two features of timed transitions, which cause an exponentially distributed firing delay, and competing immediate transitions with probabilistic firing weights. Finally, Queuing Petri Nets (QPNs) integrate and extend CPNs and GSPNs. They provide timed queuing places, consisting of a queue and a depository, to simplify modeling of scheduling disciplines [Bause and Kritzinger, 2002].

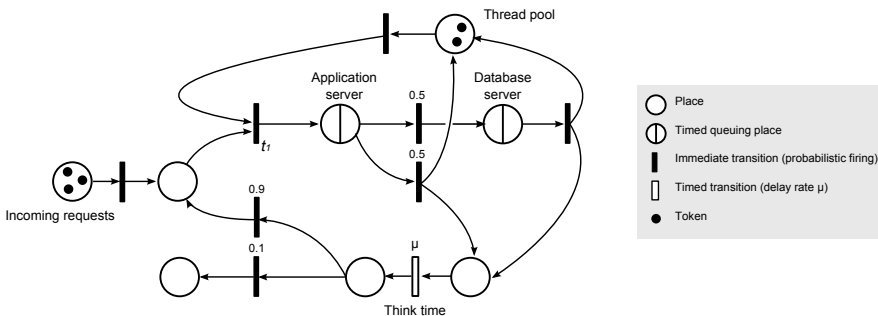


Figure 2.13. Queuing Petri Net

Figure 2.13 shows an example QPN. The example displays the same scenario as for Queuing Networks above: Requests are to be processed by an application server and a database server, with both being represented as timed queuing places. Their concrete scheduling disciplines are not specified and visualized, e.g. $G/M/1 - PS$ could be a realistic assumption. Before being processed, a request has to acquire a thread from a thread pool. The necessary synchronization is modeled at transition t_1 , which fires only if a request token as well as a thread token are available.

Performance figures are derived from Petri Nets either by analytical solutions or simulation. Analytical solutions construct a reachability graph, with each possible marking constituting a system state. In connection with the firing rates of timed transitions and probabilistic transition routes, such a Petri Net representation can be transformed to and solved as a Markov

2. Foundations

chain. As analyses based on Markov chains easily suffer from state space explosion, the complexity of the model to be solved is limited. Therefore, a practical approach is to conduct a structured top-down analysis by means of hierarchical layered models. Alternatively, Petri Nets can be simulated.

In comparison to the analysis-oriented models discussed so far, Grassi et al. [2007a] distinguish design-oriented models, which address system composition enriched with performance related attributes. Such component-based performance models and related transformation approaches are presented in the following. The requirements for these modeling approaches (including schedulable and limited resource demands, control flow, required service calls, parameter dependencies, and internal state) are discussed in more detail in [Koziolek, 2010].

The design-oriented models include OMG standardized UML profiles such as SPT, MARTE, and QFTP, as well as research approaches like the Core Scenario Model (CSM), KLAPER, and the Palladio Component Model (PCM). In some of these approaches, the idea is to start the performance analysis process on the basis of an available generic model of the software system to be analyzed. The generic model is enriched with performance related information. Afterwards, the obtained design-oriented performance model is transformed to an analysis-oriented representation for analytical solving or simulation purposes [Cortellessa et al., 2011]. This model-based performance analysis process is sketched in Figure 2.14.

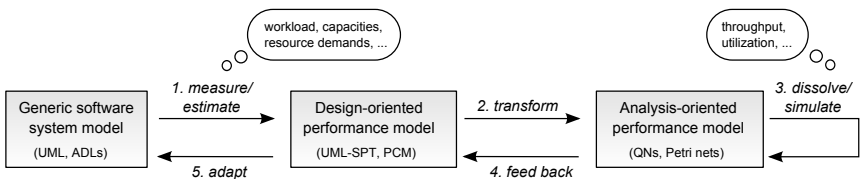


Figure 2.14. Performance analysis process

UML Profiles for Performance Modeling

The UML Profile for Schedulability, Performance and Time (UML-SPT) [OMG, 2005] is a widespread mechanism to add performance concerns to UML models. It benefits from the situation that most software designers have been educated to specify software systems using UML diagrams. UML-SPT is based on the concept to reuse existing UML diagrams (especially sequence, activity, and deployment diagrams) and to enrich them with performance related annotations. The annotated UML model can be considered as a design-oriented performance model, which will be transformed to QNs or Petri Nets for analysis purposes. After solving or simulating these underlying analysis-oriented models, the received results for performance metrics such as throughput or utilization can be feed back to the UML-SPT model.

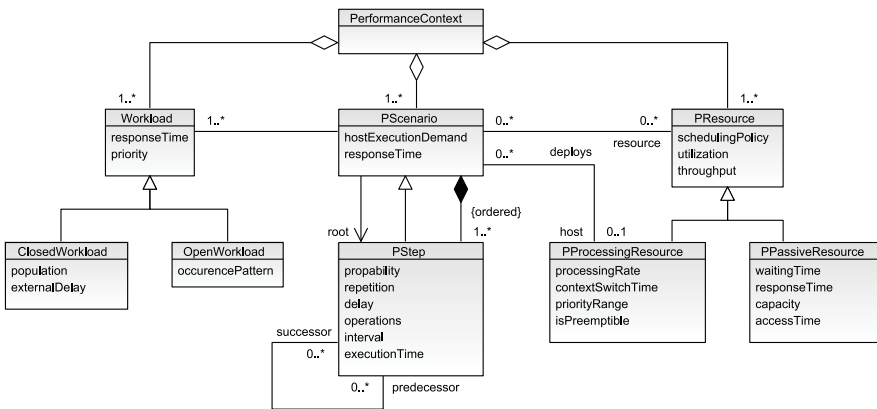


Figure 2.15. UML-SPT performance analysis meta-model [OMG, 2005]

UML-SPT consists of three domain meta-models for schedulability, performance analysis, and real-time concerns, which are based on a common resource meta-model. Particularly, the performance analysis meta-model, depicted in Figure 2.15, is of interest in this context. The central modeling entity is a scenario, which specifies a performance critical process of the analyzed system. A scenario is made up of several steps, which can easily

2. Foundations

be described via UML sequence or activity diagrams. An example sequence diagram and an associated deployment diagram are shown in Figure 2.16. Each scenario's entry step is annotated with workload information (stereotype `PAclosedLoad` or `PAopenLoad`). If the workload is closed, the population of permanently circulating requests and the delaying think time are denoted. If the workload is open, the arrival rate of new requests has to be specified. Each step can be annotated with further relevant information, e.g. resource demand in time units (tagged value `PAdemand`), repetition count, or execution probability. Resources are either passive (stereotype `PAresource`) or processing (`PAhost`). While passive software resources are tagged with their capacity (`PAcapacity`), processing resources are tagged with their processing rate (`PARate`) and context switch time (`PActxtSwT`). For both, scheduling policies (`PASchdPolicy`) can be specified. It is allowed to embed variables and mathematical expressions in the UML-SPT annotations. Finally, the annotated information should be sufficient to conclude throughput and utilization values for the resources.

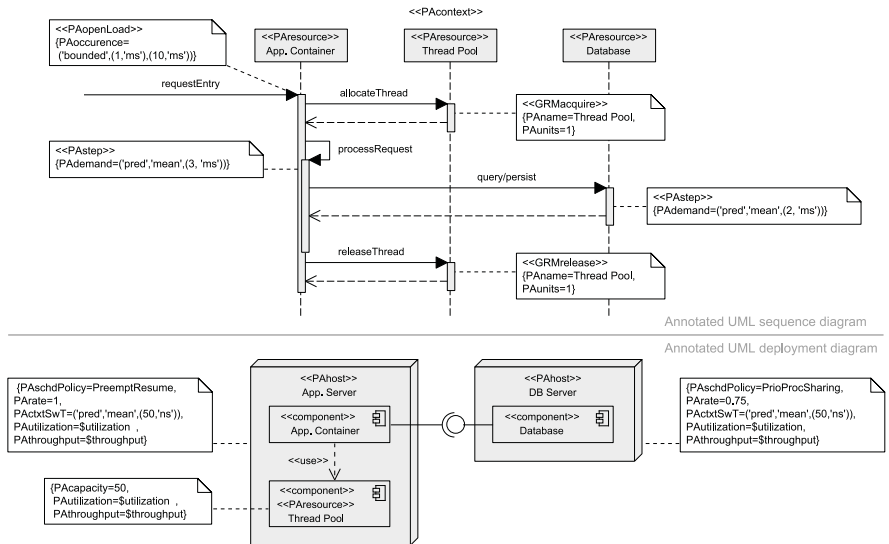


Figure 2.16. Sequence and deployment diagram annotated according to UML-SPT

A simulation approach for UML-SPT models based on transformation to QNs is provided by Marzolla [2004]. Further research approaches making use of UML profiles for performance analysis are surveyed in [Balsamo et al., 2004]. Recently, additional UML profiles have been released by the OMG. These are MARTE (Modeling and Analysis of Real-Time and Embedded systems) [OMG, 2011f] and QFTP (Profile for modeling Quality of service and Fault Tolerance) [OMG, 2008].

Core Scenario Model (CSM)

The Core Scenario Model [Woodside et al., 2005] is an intermediate meta-model for the transformation from UML design models to analysis-oriented performance models.

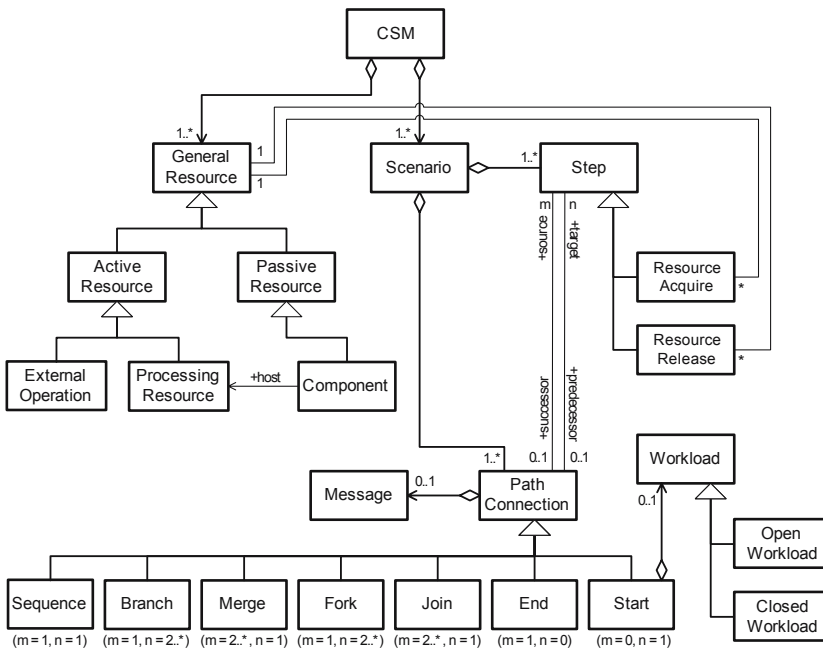


Figure 2.17. Core Scenario Model (CSM) meta-model [Petriu and Woodside, 2007]

2. Foundations

As UML-SPT annotations are spread over multiple UML diagrams (instances of sequence-, activity-, communication-, deployment diagrams, etc.), CSM aims at collecting and centralizing the information relevant for performance analysis in one intermediate model. Instead of having to define transformation rules for each complex UML diagram type, only a holistic model-to-model transformation between CSM and e.g. QNs or Petri Nets has to be developed. Consequently, CSM embodies control flow entities such as sequences, branches, and forks, so that behavior modeled in UML diagrams can be extracted. In the CSM meta-model, scenarios assembled from steps, workload, and resources are equivalent to the UML-SPT performance analysis meta-model (compare Figures 2.17 and 2.15). Resource acquisition and release are modeled as specific steps. In Figure 2.17, class attributes are left out for concision, find them in [Petriu and Woodside, 2007].

KLAPER

Another mediating approach is KLAPER (Kernel Language for Performance and Reliability analysis) [Grassi et al., 2007a,b]. Similar to CSM, it specifies an intermediate meta-model to transform any design model n to an analysis model m . Making use of such an intermediate language reduces the number of required transformation sets between possible input and output models from $n * m$ to $n + m$. In contrast to CSM, KLAPER does not provide a graphical notation. Thus, it only targets modeling tools and is not conceived as an input model for performance engineers. With KLAPER, a software system is modeled as an assembly of interacting resources. A resource is an abstract entity that can either represent a software component or a hardware resource. This universal resource concept dissents from the distinction of active and passive resources in the previous approaches. Each resource can provide and require services. The internal behavior of a service is captured by activities such as service calls, resource acquisitions, and releases. The activities are interconnected through control flow entities including branch, fork, and join.

Palladio Component Model (PCM)

The Palladio Component Model (PCM) [Becker et al., 2009] is a domain specific modeling language to describe component-based software architectures. Its major aim is to enable performance predictions for software architectures at design time. PCM is aligned with a dedicated component-based software development process, which distinguishes between the four roles of developers, architects, deployers, and domain experts (see Section 2.1.2). Each role has a limited view on the entire system model and contributes within its responsibility only specific parts to this holistic model. As illustrated in Figure 2.18, a component developer provides component specifications, a system architect constructs an assembly model, a system deployer prepares an allocation model, and a domain expert creates a usage model describing the users' behavior. The underlying meta-models for these four model types are explicated in detail in [Becker, 2008; Koziolok, 2008].

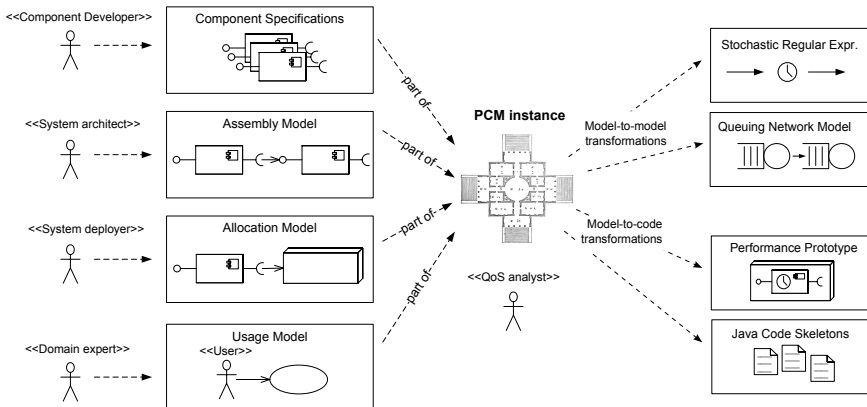


Figure 2.18. Development process of the Palladio Component Model [Koziolok et al., 2008]

PCM introduces resource demanding service effect specifications (RDSEFFs) to describe how a component's provided service utilizes resources and calls other required services. Such a specification makes the relationship between

2. Foundations

resource usage and the input arguments of a service explicit. The resource demand of component-internal algorithms is abstracted, e.g. by CPU units needed or bytes written to hard disk. Resource demands and external service calls can be nested in control flow entities, which capture sequences, probabilistic and guarded branches, loops, and forks. Further, passive resources can be acquired and released, as well as variables can be set, passed and evaluated. The notation of all PCM models is strongly inspired by UML diagrams, in order to increase the developers' learning curve and acceptance. For example, Figure 2.19 shows that RDSEFFs are visualized quite similar to activity diagrams. The depicted example RDSEFF is a representation of the code fragment in Listing 2.1. The example includes invocations of an internal CPU-demanding computation block and two external services. The first invocations are nested in a conditional branch, while the last one is repeated in an iteration loop.

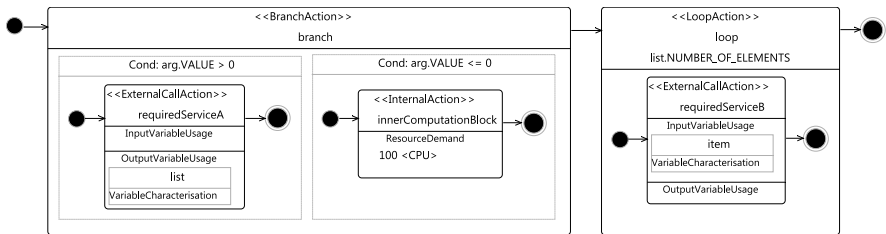


Figure 2.19. Palladio Component Model RDSEFF example

```
void providedService(int arg, List list) {
    if (arg > 0) list = requiredServiceA();
    else innerComputationBlock();
    for (Object item : list) requiredServiceB(item);
}
```

Listing 2.1. Corresponding code fragment to Figure 2.19

A benefit of PCM is its mature tool support. It comes along with an integrated modeling environment based on the Eclipse Rich Client

Platform (RCP)¹⁰ and the Eclipse Modeling Framework (EMF)¹¹. The PCM tool provides a graphical editor to create model instances, as well as integrated performance predictions and transformations. An analytical solver based on stochastic regular expressions is capable of solving single-user scenarios. Sophisticated workload scenarios can be simulated with an internal simulation framework, which is based on the discrete-event simulation library Desmo-J by Page and Kreuzer [2005]. The results of the performance predictions are visualized as histograms or cumulative distribution functions making use of the charting library JFreeChart¹². An interface to the statistics tool set R¹³ is adopted for statistical tests to validate prediction results against measurement data, if available. Furthermore, model-to-code transformations can generate a deployable prototype implementation or customizable code skeletons. An example PCM case study can be found in [Becker et al., 2007].

2.2.5 Software Performance Measurement

The complexity of enterprise software systems makes it extremely difficult to ascertain reliable predictions on the system performance at design time. In performance prediction scenarios, many rough assumptions on the timing behavior of components and the expected workload have to be accepted. By consequence, a profound insight into the effective behavior of a software system is only possible by means of runtime measurement.

Performance measurement includes the observation and collection of quantitative, particularly time-related characteristics of a system being subject to specific workload. Concerning software systems, performance measurements being part of dynamic program analysis can be distinguished between profiling and monitoring activities. Profiling activities are employed when a software system is under development in a test

¹⁰ http://wiki.eclipse.org/Rich_Client_Platform

¹¹ <http://www.eclipse.org/modeling/emf/>

¹² <http://www.jfree.org/jfreechart/>

¹³ <http://www.r-project.org/>

2. Foundations

environment. In contrast, monitoring activities take place during the continuous operation of a system.

Profiling: Profiling provides means of quality assurance intended to establish confidence that the performance requirements will be met when the system becomes productive in future. Besides the complexity of component-internal algorithms, concurrency and memory management have a significant influence on performance. In order to test the runtime behavior of a software system, developers apply profiler tools. These profilers allow the observation of synchronization and locking spots by evaluating each thread's state changes (running, waiting, blocked, etc.). In addition, runtime stack and heap states can be captured. Enabling a profiler to take regular and comprehensive snapshots of the complete system state causes intense performance overhead that is not acceptable beyond a testbed. This includes particularly allocation and reachability trees of object references on the heap, as well as related garbage collection activities. To support the comprehension of memory leaks that have occurred during productive operation, most profiler tools are able to import previously taken heap dumps. These might be produced in an operational system in case of urgent errors, e.g. if a Java Virtual Machine (JVM) instance runs out of memory. Regarding the responsiveness of operations, profilers allow the separation of clock time from CPU-service time. While clock time is the measured duration an operation needs to respond, CPU-service time is only the part of it, in which a CPU resource is effectively utilized. The rest of the clock time is spent to wait for locked resources or synchronization. Most profilers make use of byte code instrumentation (BCI) to inject probes that trigger the snapshots and record the measured data (see Section 2.3.1). A probe can be positioned at each join point, i.e. a dedicated point in the control flow, at which an observation of the current state is of interest for analysis.

For Java applications being executed in a JVM, the recently most common profiler tools include Eclipse Test & Performance Tools Platform (TPTP)¹⁴,

¹⁴ <http://www.eclipse.org/tptp/>

2.2. Software Performance

NetBeans Profiler¹⁵, JProbe¹⁶, and JProfiler¹⁷. While the first two ones are open source plugins for the respective integrated development environment (IDE), the last ones are commercial products. To inspect the state of a JVM and to control its internally executing threads, these profilers make use of a common native programming interface called JVM Tool Interface (JVM TI)¹⁸.

A recent evaluation discussing the accuracy of different Java profilers is provided by Mytkowicz et al. [2010]. It states that many profilers violate the requirement of random sample drawing. The evaluated profilers are biased by ignoring native code invocations, and by the observer effect occurring if the profiler itself perturbs the JVM's just-in-time compilation (JIT). By consequence, the profilers partially disagree on the identity of "hot" operations, i.e. operations being of interest for performance optimizations. A performance analyst working with only one profiler tool may easily be misled.

The Performance Cockpit [Westermann et al., 2010] provides a framework for the systematic execution of performance profiling experiments. The experiment setup is modeled with regards to the underlying resource environment and load profiles of the system under test, as well as specifications of the desired measurements, analyses, and result exports. The goal is to ensure replicable profiling experiments for complex setup scenarios. A similar approach is provided by Thakkar et al. [2008].

Monitoring: Monitoring provides means of quality control and improvement. Quality control is intended to ensure that performance requirements are actually met by a system being already in operation. Quality improvement is directed towards the indication of potential enhancements. With monitoring addressing efficient and robust operation as well as maintenance of a software system, the scope of software

¹⁵ <http://profiler.netbeans.org/>

¹⁶ <http://www.quest.com/jprobe/>

¹⁷ <http://www.ej-technologies.com/products/jprofiler/overview.html>

¹⁸ <http://download.oracle.com/javase/7/docs/technotes/guides/jvmti/>

2. Foundations

engineering, being primarily concentrated on the construction and evolution of software systems, is broadened. Monitoring aims at

- evidence that contractually specified SLAs are fulfilled,
- early detection of QoS problems such as performance degradation,
- supply of utilization data for efficient and dynamic resource capacity planning as in Cloud Computing environments,
- and recognition of usage patterns for interface design and marketing concerns.

As for profiling, the software system to be monitored has to be instrumented with probes. Due to the need of selecting a subset of all possible probes to be active, a monitoring model has to prescribe what, when, and where data is acquired.

What the probes ought to measure, can be denoted by defined metrics, e.g. response time in ms. A systematic process to discover the relevant metrics is the goal-question-metric (GQM) approach introduced by Basili [1992].

When the probes become active, strongly influences the sampling rate. Probe activation can either be triggered by events or predetermined time interrupts. Typically, it is desirable to draw random samples, if it is not possible to monitor and to analyze all incidents of potential interest.

Where the probes are placed, determines the origin and the feasible precision of measurements. A software system can be considered as a huge state machine, with the total state being captured by all memory values in the underlying hardware (including persistent mass storage, main memory, CPU caches and registers). Events cause system changes, e.g. an object allocation reduces the amount of free memory, or a completed instruction modifies the processor pipeline. Probes that recognize such events can either be realized by hardware or software sensors.

Hardware sensors measure voltages, temperatures, or other device activities such as fan speed. A main advantage of hardware sensors is that they do not cause overhead, i.e. they do not affect the performance of the monitored

2.2. Software Performance

system. Of special interest are so called hardware performance counters. These are special registers built into today's microprocessors being able to sense and to count predefined signals related to CPU instructions and cache or memory operations. However, correlating the low-level data collected by hardware performance counters to running applications that the users interact with remains challenging [Sweeney et al., 2004; Ammons et al., 1997; Schneider et al., 2007].

Software sensors consist of routines inserted into different software layers. Today's, enterprise software systems stretch across several virtualization layers, beginning at the application level down to the hardware resource level (see Figure 2.20). The virtualization layers include operating systems, runtime environments such as the JVM and the .NET Common Language Runtime (CLR), and middleware such as Java EE application servers. Monitoring routines can gather performance characteristics about executing processes at all these layers.







Application	
Framework, libraries	
Middleware application server	
Virtual machine runtime	
Native libraries	
Operating system	
Hypervisor	
Hardware resources	

Figure 2.20. Software system virtualization layers

Compared to languages that are immediately compiled to machine code such as C and C++, an intermediate runtime environment like a JVM that runs managed byte code provides more flexibility and security at runtime. This includes features like reflection, automatic memory management, security policies, and runtime exception checking (e.g. for null pointers or array bounds). In addition, the virtual machine byte code is portable

2. Foundations

across different computer architectures. Unfortunately, the additional layer of virtualization introduces obstacles to application-level performance comprehension. Benchmarking code managed by a JVM is complex, as the behavior of a Java application is significantly influenced by dynamic recompilation (JIT), garbage collection, and thread scheduling. Dynamic recompilation leads to nondeterminism, with one source code statement being possibly transformed to changing machine code instructions operating at different memory locations throughout runtime. A technique directed towards controlling nondeterminism is replay compilation. The idea of replay compilation is to compensate the compilation overhead by dictating a previously recorded compilation plan each time an experiment is replicated. This improves the experimental repeatability. Further recommendations for statistically rigorous performance evaluations of Java applications are discussed by Georges et al. [2007].

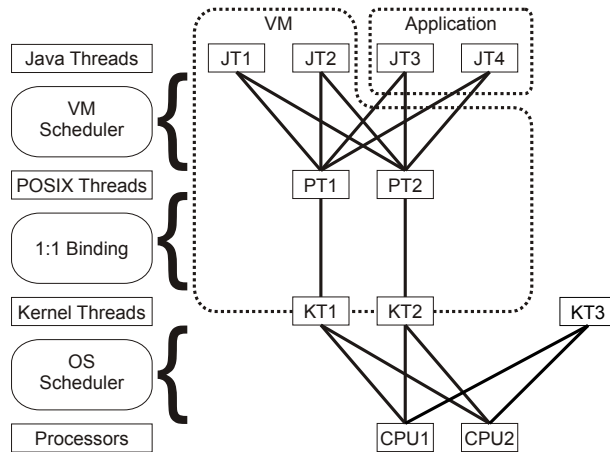


Figure 2.21. Jikes RVM scheduling [Hauswirth et al., 2004]

The vertical correlation of hardware and software sensor data over multiple virtualization layers is complicated. A first approach to trace the effect of application-level source code statements down to hardware layer is provided by Hauswirth et al. [2004]. The authors concentrate on monitoring

2.2. Software Performance

components of the JVM such as the memory manager, the runtime compiler, and the synchronization control. Furthermore, they record system call and signal interactions between the virtual machine and the operating system. Making use of the Jikes Research Virtual Machine (Jikes RVM)¹⁹ by Alpern et al. [2005], it is possible to map Java threads across POSIX threads to OS kernel threads. The scheduling in the Jikes RVM is illustrated in Figure 2.21. To determine the origins of performance phenomena, performance metrics and source code have to be correlated, e.g. it might be shown that a specific source code statement causes many cache misses.

As application-level monitoring is not directed towards the primary application goals, it is a non-functional cross-cutting concern. The instrumentation of the monitoring concern, i.e. the used technique of integrating the monitoring capabilities, should be non-intrusive, so that any code implementing the application's business logic has not to be modified when monitoring is (re)configured. In legacy software systems, monitoring is often appended manually and mixed with the business logic on code level. This violation of concern separation downgrades the readability and maintainability of the source code. Techniques for non-intrusive instrumentation are middleware interception and source code extensions by means of aspect-oriented programming (AOP) or BCI. Probes are injected at the entry and exit points of component-internal operations in order to measure response times. Typically, it is not feasible to take continuous snapshots of the complete system state including the heap. Instead, probe filters determine which parts of the control flow, e.g. method invocations, and data flow, e.g. method input arguments, are monitored.

In case of monitoring a distributed system, a *monitoring agent* is required for each execution container of the application code. The monitoring agents report to a central instance where their collected data is merged and analyzed (see Chapter 4). Regarding continuous operation, gradual aggregation of monitoring data over time becomes necessary, because the collected data volumes quickly exhaust storage otherwise. Additionally,

¹⁹ <http://jikesrvm.org/>

2. Foundations

fine-grained monitoring data over a number of months or years is usually irrelevant for analysis.

2.3 Instrumentation

In the following different instrumentation techniques to inject application-level monitoring are discussed. These techniques are general and can be applied to any virtual machine. Concrete implementations are discussed for the JVM, being regarded as the most prominent virtual machine today.

2.3.1 Byte Code Instrumentation (BCI)

A major goal during the design of Java was the portability of applications across different computer architectures. This has been achieved by introducing the virtual machine concept. Each JVM instance is translated to the machine code of the underlying platform, while itself runs intermediate Java byte code, being inclosed in a set of class files. A Java compiler generates for each source code file (.java) a class file containing the corresponding part of byte code (.class). A class file is structured by the following order of sections: magic number, version, constant pool, access flags, this class, super class, interfaces, fields, methods, and meta-attributes [Lindholm and Yellin, 1999]. Sections such as fields and methods that are of variable length are prefaced with their actual byte length. BCI aims at modifying class files before they get completely loaded into the JVM. The JVM is specified to be a stack machine, maintaining a stack for each thread and a heap and a method area being shared among all threads. If a class is addressed for the first time, it is loaded by a class loader, which instantiates the class structure in the JVM's method area. Thus at runtime, the method area contains the structural composition of each loaded class, i.e. its constants, fields, and methods including each method's sequence

2.3. Instrumentation

of byte code instructions. The virtual machine's byte code instruction set can be grouped into stack operations, arithmetic operations, control flow, load and store operations, field access, method invocation, object allocation, and conversion and type checking. There are two different approaches to invoke instrumentation. The first approach is to read a class file directly after compilation and to replace it with a new file version, which includes the intended modifications. The second approach is to instrument a class as recently as it is loaded. Advantages of this approach are the omission of checks whether class files have already been altered and that the original file version is not overwritten. To realize load-time instrumentation, one can either use a proprietary class loader that extends the JVM's default class loader or register an agent to instrument the byte code. The usage of self-defined class loaders can interfere with frameworks or application servers that introduce their own class loaders.

```
public class MyInstrumentationAgent {

    public static void premain(String options, Instrumentation instr) {
        instr.addTransformer(new ClassFileTransformer() {

            @Override
            public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
                ProtectionDomain protectionDomain, byte[] classfileBuffer) throws IllegalClassFormatException {

                // Instrument the current class file byte code in <classfileBuffer> ...
                return MyBCILibrary.instrument(classfileBuffer);
            }
        });
    }
}
```

Listing 2.2. Load-time byte code instrumentation by attaching a Java agent

A Java agent is an archive that manifests a class with a static method `premain` according to the signature shown in Listing 2.2. As indicated by its name, this method is executed previously to the main method of any started application that the agent is attached to. Instrumentation can be enabled via the second input argument, which is an instance of `java.lang.instrument.Instrumentation`. It allows registering class file transformers, implementing the interface `java.lang.instrument.ClassFileTransformer`, as demonstrated in Listing 2.2. A

2. Foundations

registered transformer's method transform will be called each time a class is loaded and can imply goal-oriented changes of the class structure or its method-internal behavior.

As it is not trivial to manipulate the byte code and gain a valid outcome afterwards, libraries have been established that simplify the modification process. These include the Byte Code Engineering Library (BCEL)²⁰ introduced by Dahm [2001], ASM²¹ [Bruneton et al., 2002], and Javassist²² [Chiba, 2000]. Javassist offers the most user-friendly API, with equivalent performance to BCEL [Sosnoski, 2004]. An example code fragment that demonstrates how to add a logging statement to an arbitrary class method is given in Listing 2.3. After identifying the given method object, the insertBefore method compiles the specified statement and inserts it at the beginning of the method body.

```
void instrument(String className, String methodName)
    throws javassist.NotFoundException, javassist.CannotCompileException {
    javassist.ClassPool cp = javassist.ClassPool.getDefault();
    javassist.CtClass c = cp.get(className);
    javassist.CtMethod m = c.getDeclaredMethod(methodName);
    m.insertBefore(
        "System.out.println(\"Method \" + methodName + \" in class \" + className + \" has been invoked.\");");
    }
}
```

Listing 2.3. Byte code modification example using the Javassist API

In contrast, the benefits of BCEL are its extensive support at the level of JVM byte code instructions, its longevity, and the resulting stability. Further, the Apache licensing of BCEL (Apache License 2.0) is more appropriate for commercial exploitation than the Javassist license conditions, which are to satisfy either the Mozilla Public License (MPL) or the GNU Lesser General Public License (LGPL).

²⁰ <http://jakarta.apache.org/bcel/>

²¹ <http://asm.ow2.org/>

²² <http://www.jboss.org/javassist/>

2.3.2 Aspect-Oriented Programming (AOP)

The AOP paradigm introduced by Kiczales et al. [1997] particularly addresses secondary concerns like monitoring that cut across the basic functionality of a system. Instead of scattering these cross-cutting concerns throughout the code, they are isolated in so called aspects to increase modularity. The most popular Java-based AOP implementation is AspectJ²³ [Kiczales et al., 2001]. Originally, it was developed at the Palo Alto Research Center (PARC), formerly belonging to Xerox. When AspectJ was contributed to the Eclipse open source projects, it has been reimplemented using an enhanced bytecode weaver based on BCEL. Meanwhile, AspectJ has been integrated into the widespread Spring Framework [Laddad and Johnson, 2009; Spring Source, 2011]. Another AOP implementation is GluonJ [Chiba and Ishikawa, 2005], which is built on the Javassist BCI library and claims to be more suitable for expressing inter-component dependencies than AspectJ. Along with AspectJ, the following universal terms have been coined.

- *Aspect*: An aspect is a modularization type for a cross-cutting concern. It consists of advices on pointcuts that laterally introduce new behavior into other types. Like a class, an aspect can declare internal data structures and behavior, can extend other types, and can implement interfaces. Examples for aspects are transaction management, authorization control, and last but not least performance monitoring.
- *Join point*: A join point is a well-defined point in the execution of a program, e.g. the execution of a method, the construction of a class instance, or the handling of an exception.
- *Pointcut*: A pointcut is a set of join points expressed by a predicate that matches the join points to be selected, e.g. all method calls within a class or a package.
- *Advice*: Being linked to a pointcut expression, an advice specifies an operation that runs at any join point matched by the associated

²³ <http://www.eclipse.org/aspectj/>

2. Foundations

pointcut. Additionally, an advice defines whether it is executed before, after, or around the affected join points.

There exist different styles to declare aspects in Java. These include (1) the original AspectJ language code style, (2) aspect definition based on XML files (Spring AOP style), and (3) aspect definition via annotations (available since Java 5). Listing 2.4 makes use of the annotation style. The listing demonstrates how a logging statement can be added to all methods that are contained in a package being matched by the specified pointcut expression.

```
@Aspect
public class MyLoggingAspect {

    @Pointcut("execution(* com.example..*(..))")
    public void pointcut() {
    }

    @Before("pointcut()")
    public void interceptedMethod(JoinPoint jp) throws Throwable {
        System.out.println("Join point " + jp + " reached at " + new Date());
    }
}
```

Listing 2.4. AspectJ annotation style aspect definition

AspectJ provides several key functions, such as `execute` as used in Listing 2.4, that can be used in combination with typed elements, modifiers, wildcards, and logical operators to formulate pointcut expressions. For example, these functions include the following ones that trigger under specific conditions: `execute` when a particular method body executes, `call` when a method is called (from view of the enclosing caller site), `handler` when an exception handler is invoked, or `this` when the currently executing object is of a specified type.

The process of weaving aspects into the class files can either take place at compile time or at load time (see Figure 2.22). With compile-time weaving, the aspects are woven into the byte code just after or during the compilation process. Load-time weaving defers the weaving until a class is loaded into the JVM. To support this, either a special-purpose class loader is required or class loading is intercepted by a weaving agent. Weaving at runtime

2.3. Instrumentation

without class reloading is not supported by AspectJ. Though, enabling and disabling advices is realizable programmatically.

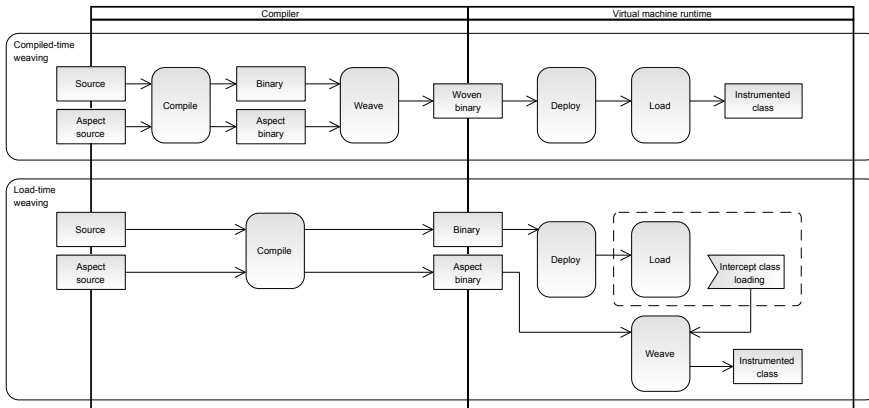


Figure 2.22. Comparison of compile-time and load-time weaving

2.3.3 Middleware Interception

The idea of middleware interception is to observe component-level interaction by means of the underlying middleware. Probes are attached as component wrappers that insert small proxy elements to each component. For example, the COMPAS Java End-to-End Monitoring (JEEM) tool leverages the provided meta-data of Java EE components to automatically generate such wrappers [Parsons et al., 2006]. Instrumentation takes place at deployment time. Concerning the archives that manifest the target components like enterprise archives (.ear) or web archives (.war), the inherent meta-data and deployment descriptors are evaluated in order to extract and manipulate the component interfaces. The probes are inserted as proxies inspecting and delegating calls to the actual components, e.g. to EJBs at the business tier, or to Servlets at the web tier. The approach is non-intrusive and portable across any Java EE application server, but interception is limited to the granularity of component interfaces.

2. Foundations

Each Java EE application server itself comes along with some basic monitoring features related to managed resources, such as thread and database connection pools, or specific Java APIs. Typically, API class instances are introduced into applications making use of the dependency injection pattern. Thereby, the container framework keeps the ability to control and to monitor the application's control flow at specific points. This includes commonly used features such as transactions, persistence, or inter-component service calls and communication related to the following APIs²⁴:

- Java Transaction API (JTA) as in JSR 907
- Java Persistence API (JPA) as in JSR 317
- Enterprise JavaBeans 3.1 (EJBs) as in JSR 318
- Java API for XML-based Web Services (JAX-WS) as in JSR 224
- Java API for RESTful Web Services (JAX-RS) as in JSR 311
- Java Message Service (JMS) as in JSR 914

The measures collected by the container are accessible via the Java Management Extensions (JMX) interface²⁵ [Sun Microsystems, Inc., 2006]. Other instrumentation standards such as Application Response Measurement (ARM)²⁶ [Johnson, 1998] or finer application-specific instrumentation with JMX rely on the compliance to the provided API. By consequence, the decision to use such a performance management standard has preferably been taken at design time. It is more difficult to instrument existing systems because code modification at various locations become necessary.

²⁴ Find a list of all Java specification requests (JSRs) at <http://www.jcp.org/jsr/all>.

²⁵ <http://download.oracle.com/javase/7/docs/technotes/guides/jmx/>

²⁶ <http://www.opengroup.org/tech/management/arm/>

2.4 Self-Adaptive Software Systems

Adaptability tends to become a key feature of long-living software systems, which have to cope with the requirement to adjust their architecture or behavior on demand. The evolution of operational contexts and execution environments causes modern software systems to be highly versatile and resilient towards changing requirements. At the same time, as our society gets more and more dependable on software-supported services, the hazards, inconveniences, and costs related with potential downtimes of these services increase. Time-consuming restarts of operating systems or mission-critical applications associated with patches or upgrades are not tolerable anymore, not only for energy providers and financial institutes. Today it is common, that availability rates are contractually specified via SLAs between hosting providers and their business clients. Thus, the requirement arises that such software systems have to be adaptable at runtime.

Beyond that, it might be desirable that the system itself is capable of reacting on changing demands and finally adapting its own composition or configuration. This is the case, if frequent fine-grained adaptation decisions are required to operate the system as efficient as possible. For example, if energy-efficient utilization of the available resource capacities is aspired, frequent adaptations of load balancing, component replication and deployment are required. These can be executed by a self-adaptive system without human interaction. The vision is a software system, that is capable of self-management, self-adaptation, self-configuration, self-healing, self-tuning, and further semantically similar self-* terms [Salehie and Tahvildari, 2009]. Research in the software engineering community has been inspired by related fields, such as robotics and artificial intelligence, as well as other research domains, particularly biology.

The approach proposed in this thesis aims at runtime adaptability of software systems at application level. During the last decade, the following lower-level adaptation techniques have been developed from

2. Foundations

research prototypes to industrial marketable products. In many scenarios, architectural and behavioral software adaptation solutions are built upon these techniques as basic prerequisites [Oreizy et al., 2008]:

- Hot-plugging of hardware devices, such as disk drives or memory chips, to increase capacity or replace defective devices without switching off power.
- Tuning of operating system, driver, or firmware parameters to seek for a more efficient resource utilization (e.g. for CPUs) without the need to reboot.
- Runtime adaptable hardware virtualization (e.g. with tools like VMWare²⁷, Xen²⁸, or Microsoft Hyper-V²⁹) to dissolve the fixed assignment between hardware resources and operating systems. Again, the major objective is a higher flexibility of capacity allocation to improve resource utilization.
- Hot code replacing as a feature of programming language runtime environments (e.g. JVM, .NET CLR) to load, check, and invoke new code fragments at runtime.

Self-adaptive software control is investigated by various research communities, with each one being motivated by a specific application domain, such as peer-to-peer systems, multi-agent markets, mobile and autonomous robots, ubiquitous computing, embedded systems, ad hoc networks, and self-learning user interfaces [Cheng et al., 2008]. The application domain addressed throughout this thesis, especially in Chapter 3, is the monitoring process of component-based enterprise software systems. It is notable that the common ground providing self-adaptation in all applications listed above is a piece of software.

²⁷ <http://www.vmware.com/>

²⁸ <http://www.xen.org/>

²⁹ <http://www.microsoft.com/virtualization/>

2.4.1 Classification Dimensions for Self-Adaptive Systems

Self-adaptive systems can be classified with respect to characteristic dimensions. The dimensions outlined in this section are merged from the classification approaches of Cheng et al. [2008] and Rohr et al. [2006]. Each dimension explicates a relevant aspect of self-adaptation requiring an explicit design decision while system modeling.

Supervision: Supervision characterizes the form of organization chosen to realize self-adaptation. In particular, it refers to the placement of the adaptation controller. It can be either centralized (weakly organized), decentralized (strongly organized), or a hybrid mixture [Di Marzo Serugendo et al., 2007].

In case of a centralized supervision, the adaptation controller is tightened in a single entity being in charge of decision-making how and when the system's components have to adapt their behavior or interaction. Typically, there is only one major control feedback loop that processes sensor data and triggers the actuators to adapt the system (see Section 2.4.2). It has to be taken into consideration that a centralized controller easily tends to be a bottleneck and a single point of failure.

In case of a decentralized supervision, there is no global view of the system. Each component maintains its own controller in order to adjust itself in reaction to internal or environmental sensor data. Communication and cooperation strategies in such a self-organizing topology are usually based on peer-to-peer solutions for distributed systems, e.g. as in [Chakravarti and Baumgartner, 2004]. According to the following common definition of a software agent by Wooldridge [2009], collaborative or competitive software agent systems are prominent examples for a decentralized form of supervision. Hybrid approaches try to combine the advantages of both supervision forms by controlling and executing some parts of adaptation locally and others globally.

2. Foundations

Definition: Software Agent by Wooldridge [2009]

An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.

Stimulus triggering: The stimulus dimension characterizes the possible sources of adaptation decisions. Like in biology, a state change (stimulus) causes a reactive response, which is, in the case of self-adaptive systems, any generic adaptation operation. The source of a stimulus can either be originated from inside of a system component or from its environment. Figure 2.23 shows the conceptual separation, inspired by control theory, between a controller entity and a controlled, adaptable component. Sensors are linked to the controller to be informed about system changes of interest. In the opposite direction, the controller uses actuators to apply adaptation decisions and manipulate its controlled components. Given a sophisticated system model, environmental sensor data can be examined in more detail, e.g. to analyze component-level cause-effect-chains or to see if some distinct human interaction triggered a specific adaptation. The usage of AOP to employ sensors and actuators as a cross-cutting concern targeting adaptation is exemplified in [Salehie et al., 2009; Haesevoets et al., 2010]. Further means to realize sensors and actuators are surveyed in [Salehie and Tahvildari, 2009].

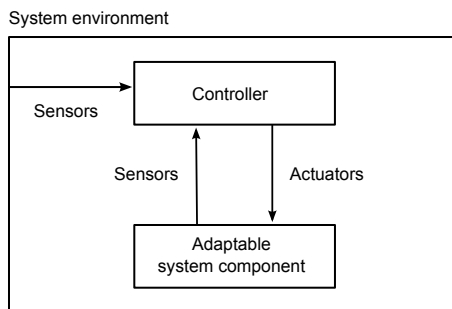


Figure 2.23. Sensors and actuators of a self-adaptive system, cf. [Rohr et al., 2006]

2.4. Self-Adaptive Software Systems

System layer: The system layer dimension refers to the level at which adaptation takes place. Commonly, at least four basic layers of a software-driven system are distinguished: hardware, operating system, middleware, and application. As self-adaptation is a general concept, it can be applied at each of these layers. Many research approaches concentrate one or two layers and act on assumptions concerning the lower or higher layers. For instance, adaptation strategies on application level can assume a specific middleware platform, which supports monitoring or management services that are employed, e.g. [Mos and Murphy, 2004]. Moreover, observation and adaptation activities can be aligned on different layers. For example, if a system's utilization is monitored on middleware or application level, but reconfiguration is accomplished by means of component redeployment on hardware or operating system level, e.g. [van Hoorn et al., 2009a].

Degree of automation: The automation dimension describes the degree of human interaction needed for adaptation. The degree ranges from autonomous to human-controlled. While perfectly self-adaptive systems do not require any human intervention, others may depend on human decision or confirmation before critical adaptation operations, or at least some kind of approval afterwards. There exists a broad range of techniques for machine-based decision-making such as utility functions, case-based reasoning, neural networks and other soft computing methods, statistical pattern recognition for classification or regression, etc., see [Bishop, 2006; Duda et al., 2000].

Type of adaptation operations: The different types of adaptation operations range from smart parametric reconfigurations to architectural recomposition activities. That is, self-adaptation can be embodied by the variation of local parameters that change the internal behavior of a component. As well, it can involve comprehensive compositional modifications that change the entire system's assembly or deployment. A classification scheme for adaptation operations from an architectural viewpoint is provided by Rohr et al. [2006], based on preliminary work in [McKinley et al., 2004; Cuesta et al., 2001]:

2. Foundations

- Data flow: Values of the system data model are modified that only influence data flow, but do not effect the control flow inside or among the system’s components.
- Intra-component behavior: The internal behavior of a component, i.e. its control flow or quality of service, is changed without modifying any sequences of inter-component service calls (compare to component interaction below). Although the implementation of a component instance changes, its associated component type, indicated by the provided and required interfaces, is not affected.
- Component resource mapping: The deployment of components regarding their underlying physical or virtual execution environment is modified at runtime.
- Component interaction: The communication between the existing components is adapted by changing sequences of inter-component service calls.
- Component instances: The system’s structural assembly is changed by adding new component instances from a fixed pool of known component types, or by (re)moving present component instances. The system assembly can contain multiple component instances, which are of the same type.
- Component types: Adaptation operations allow the definition of new component types dynamically. By consequence, the system architecture is completely mutable at runtime.

This classification scheme refers to adaptation operations at design level. Besides, it can be looked at how adaptability impacts the artifacts at other common levels of abstraction. For example, artifact code at implementation level can be adapted by dynamic weaving of aspects at compile- or load time, alternatively by hot code replacement at runtime. A future challenge is the reasonable adaptation of fundamental goals defined at requirement level.

Figure 2.24 shows a categorization of adaptive application types by McKinley et al. [2004], which is determined by the time at which the adaptation takes place. This classification scheme significantly depends on

2.4. Self-Adaptive Software Systems

the adaptation technique that enables the desired compositional adaptation. A corresponding catalog of techniques is embodied at the bottom of Figure 2.24, as well.

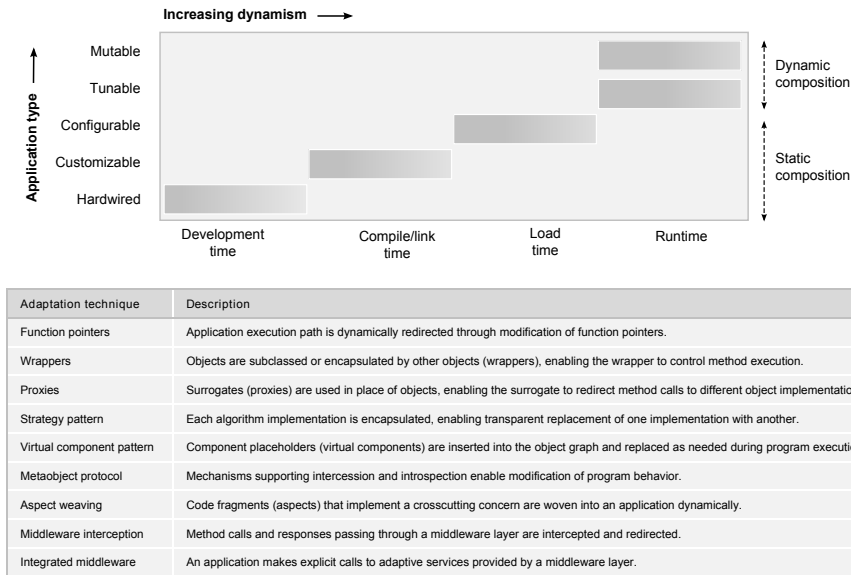


Figure 2.24. Adaptation time, types, and techniques [McKinley et al., 2004]

Timing: The timing dimension is related to self-adaptation timing issues such as internal responsiveness and potential impacts on the system performance.

Responsiveness is aligned to superior activation guidelines. The desired type of activation can be reactive, predictive, or proactive. The semantic distinction between these three attributes is illustrated using the performance development of a software service as an evident example:

- *Reactive:* The type of adaptation activation is reactive, if the system adapts after the monitored performance drops under a specified

2. Foundations

threshold. Thus, the performance degradation may be externally perceivable.

- *Predictive*: The activation type is predictive, if the system adapts as a reaction on alarming events or states that indicate a future drop of performance, before the actual perceivable drop occurs to the system users.
- *Proactive*: The activation type is proactive, if the system is capable of goal-oriented self-adaptation that tunes the system's performance. It is continuously tried to optimize the performance, although it is already at a satisfying level and does not tend to decline.

Another aspect of timing describes the degree of temporal determinability regarding self-adaptation response to changes of the system state. Especially in real-time scenarios, e.g. in automotive airbag or braking systems, the responsiveness has to be guaranteed. In less critical scenarios, best-effort approaches are sufficient.

As self-adaptation itself requires processing time, it can have an impact on the performance of the adaptable system, which it is part of and shares resources with. That monitoring biases an observed phenomenon, is well-known as the observer effect. A quantitative evaluation concerning performance analysis is provided by Mytkowicz et al. [2008]. In time-critical cases, the overhead induced by self-adaptation has to be predictable. A relevant factor is how frequently adaptation operations, respectively complex preceding analyses are processed. The process triggering can be either event-based, when a significant system change is transmitted via sensor data, or time-based, when it is initiated at predetermined points in time.

Dependability: Finally, different quality criteria concerning the dependability of self-adaptation can be surveyed. The dependability of a system is defined as the ability to deliver service that can justifiably be trusted. Alternatively, the justification of trust is replaced by individual perception, conceiving dependability as the ability to avoid service failures that are more frequent and more severe than is acceptable [Avizienis et al.,

2.4. Self-Adaptive Software Systems

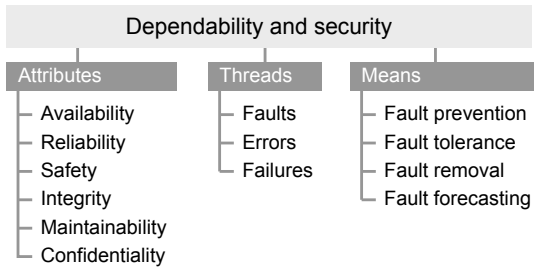


Figure 2.25. Dependability taxonomy, cf. [Avizienis et al., 2004]

2004]. Dependability covers the following attributes, being considerably significant for self-adaptive systems:

- Availability: Readiness for correct service, for authorized actions only,
- Reliability: Continuity of correct service.
- Safety: Absence of catastrophic consequences on the users and the environment.
- Integrity: Absence of improper and unauthorized system alterations, in particular data integrity concerning concurrent transactions.
- Maintainability: Ability to undergo modifications and repairs.

Confidentiality, i.e. the absence of unauthorized disclosure of information, is added as a further attribute, if security is addressed along with dependability. In this case, security would be a composite of confidentiality, integrity, and availability. Threats and means impacting the dependability attributes are summarized in Figure 2.25.

2.4.2 Control Feedback Cycle

Self-adaptive systems have in common that decisions affecting the system's composition or behavior are moved from design time towards operation time. The system itself observes and reasons about its state and environment. Observation and reasoning involve a control feedback process, which is

2. Foundations

permanently iterated. Each iteration of this cycle runs through the activities of monitoring, analysis, adaptation planning, and adaptation execution, which are described in the following. Reviewing some relevant research publications, the naming of the activities differs, as shown in Figure 2.26. Nevertheless the general understanding of the control feedback process is conform. This thesis follows the original terminology of the IBM autonomic computing community, abbreviately known as MAPE-K [Kephart and Chess, 2003]. While Dobson et al. [2006] and Garlan and Schmerl [2002] only use other terms for the activities, other authors merge the adaptation planning activity into the analysis activity, e.g. [Rohr et al., 2006], or into the adaptation execution activity, e.g. [Baresi et al., 2008]. Inspired the three-layer robotic architecture (control, sequencing, deliberation) [Gat, 1998], Kramer and Magee [2007] propose a hierarchical reference model, in which sensors and actuators are integrated to a bottom control layer serving as a common interface for the adaptable components.

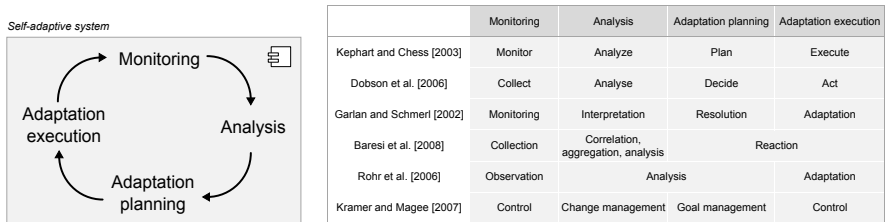


Figure 2.26. Control feedback cycle of self-adaptive systems

Müller et al. [2008] argue that the control feedback process is a crucial feature of self-adaptive systems, which should rather be elevated than hidden or abstracted, when a system’s architectural model is presented. In the following the four activities (1) monitoring, (2) analysis, (3) adaptation planning, and (4) adaptation execution are discussed in more detail.

Monitoring: The control feedback cycle starts with the collection of relevant data reflecting the current system state. This is implemented by monitoring sensors, which are attached to components and the execution environment.

2.4. Self-Adaptive Software Systems

Engineering questions that arise during the design and implementation of the monitoring activity include:

What kind of information is retrievable? What is the required level of detail and sample rate?

As monitoring is applied to operational systems, only a deliberately small performance overhead should be imposed. Usually, a trade-off has to be made between the acceptable overhead and the retrieved information quality regarding detail and sampling rate. It has to be conceived which information is really needed and what can be technically measured. Irrelevant sensor data may even cause unnecessary computational cost in the subsequent analysis activity. The requirements analysis and design of the monitoring activity should end up with a systematically modeled process that describes what, where, and when sensor data is measured (cf. Section 2.2.5).

How can monitoring capabilities be instrumented in non-intrusive ways? How should future systems and their components be designed to support easy monitoring?

Considering that monitoring is a cross-cutting concern, appropriate instrumentation techniques are either middleware interception or source code extension at compile time or load time. Compared to common logging APIs, these techniques provide the critical benefit to be non-intrusive, so that the application's business logic has not to be touched when monitoring code is modified. An isolated realization of the monitoring concern improves the system's maintainability. If a new software system is designed, the exploitation of standardized monitoring middleware interception techniques such as JMX can be recommended. Otherwise, if an existing system has to be extended by monitoring capabilities, encapsulated source code extensions like AOP or BCI are appropriate (cf. Section 2.3).

How reliable is the data delivered by the sensors?

Caused by technical barriers, the measured sensor data might be fuzzy or imprecise. Further, it is often not feasible to measure a continuous data

2. Foundations

stream, but probes provide values at dedicated points in time. Nevertheless, subsequent analyses assume a steady progression for the gaps between the measurements. It has to be questioned, if the reliability of the results is not disrupted, e.g. in case of CPU or fine-grained memory utilization measures.

Analysis: Next, the collected raw data is structured, aggregated and correlated. The analysis activities are directed towards the later decision requirements. Engineering questions include: How is the current system state inferred and modeled? How much past data has to be archived for future inferences and validations? Complex analyses may cover statistical methods from the soft computing or pattern recognition research [Salfner et al., 2010], which are exploited for reasoning during the adaptation planning activity.

Adaptation Planning: This activity covers the decision-making process how and when to adapt the system. Utility functions and case-based reasoning are applied making use of the previously established analysis data set. If frequent adaptations are not possible, risk analysis has to guarantee that no wrong or precipitate decisions are taken. How much human interaction is required, depends on the degree of automation, as described above. Engineering questions to be answered are: How to select the best alternative from a set of possible adaptations? Is it justified to act on the assumption that the scheduled adaptations will improve the system's quality? How to tune the system if no failures or anomalies are perceived?

Adaptation Execution: In many iterations of the control feedback cycle, the decision will be taken that no adaptation is required. Therefore, the adaptation execution activity is regularly skipped and carried out less often than the previous activities. The different kinds of adaptation operations, which are executed via the system's actuators, have been discussed above. The addressed engineering questions include: Can the adaptation operations always be safely performed? How does the system behave if something goes wrong during the process of adaptation? Especially, changes on architectural level are risky, as untested assembly and deployment configurations may have a significant impact on the delivered

2.4. Self-Adaptive Software Systems

QoS, and as adaptation operations itself can be technically challenging with regard to the requirement not to interrupt ongoing transactions within the operational system [Matevska, 2008].

Self-Adaptive Performance Monitoring Approach

Software engineers consider performance to be a highly critical requirement. At the same time, tools that monitor the operation of software systems at application level are rarely used in practice. This contradiction is indicated by a survey carried out at a recent conference among Java practitioners and experts [Snatzke, 2009]. Typical issues such as garbage collection, concurrency, remote service calls, especially database access and legacy integration are identified as common problem areas. Nevertheless, adequate monitoring tools that allow analyzing these problems and their root causes are sparsely known and employed in software engineering projects.

Particularly, there exists no widespread and mature *open source* solution for application performance monitoring (APM). The popular open source tool Nagios¹ is rather intended for infrastructure monitoring than for application-level introspection. Commercial APM software such as DynaTrace and AppDynamics is discussed in Section 6.2.

In today's practice, APM is typically applied in a reactive, inflexible, and decentralized way. A general undervaluation of continuous operational monitoring is expressed from the following shortcomings.

- *A posteriori failure analysis*: Monitoring data is seldom evaluated systematically before a service failure is reported.

¹ <http://www.nagios.org/>

3. Self-Adaptive Performance Monitoring Approach

- *Inflexible instrumentation*: Probes are placed into a component's code only at a limited number of fixed measuring points. Recompile and redeployment are required for future modifications.
- *Inability of tracing in distributed systems*: Tracing of user requests beyond the borders of a single component or its execution container in a distributed system is not supported or not applied.

3.1 Application Goals of Adaptive Monitoring

This thesis presents a method for self-adaptive performance monitoring and exemplifies the design of an associated tool to handle the above shortcomings for existing component-based Java EE applications. As explicated in Section 2.2.5, means for efficient and flexible observation, analysis, and reporting of a software system's runtime behavior are indispensable to ensure its proper operation. Thereby, two constraints have to be considered: (1) acceptable overhead and (2) non-intrusive instrumentation. Unlike to profiling at construction time, the monitoring overhead at operation time has to be kept deliberately small. Secondly, the instrumentation of probes should not impurify the application's business logic.

A main issue for analysis approaches addressing software behavior comprehension is the amount of information which is collected and processed at runtime. As a matter of principle, the more detailed monitoring data is, the more precise subsequent analysis can be. On the other hand, instrumentation, data collection, data logging, and online analyses cause measurable overhead. By consequence, a trade-off between analysis quality and monitoring coverage has to be reached.

It is not the injection of numerous dummy probes that causes overhead, but the complexity of real probe implementations. The decisive impact of how the observed data is collected, logged, and processed in subsequent analyses is evaluated and quantified in Section 5.3. A finding is that it is generally

3.1. Application Goals of Adaptive Monitoring

possible to inject diverse probes at a multitude of relevant measuring points, as long as not all of them are active at the same time during operation. Thus, the approach aims not at a fixed number of probes to be injected or fixed measuring points to be activated. Instead, a major objective is a self-adaptive activation of probes and their related measuring points. For that purpose, an inference procedure is needed and proposed to decide at which granularity a component should be observed. A performance engineer's task is to specify rules that reveal the major monitoring goals for a specific application. These may reflect the following generic goals.

Evidence of SLA compliance: It is required to generate evidence that contractually specified SLAs are fulfilled. The contracting client parties will probably test the service's externally perceived responsiveness and availability. In case of dispute, the service provider needs to audit the delivered QoS independently. Besides general usage conditions and contractual penalties, SLAs primarily contain measurable metrics called service level objectives (SLOs) that both sides agree on [Keller and Ludwig, 2003]. Popular metrics are response time distributions, concurrent session capacities, and availability rates. For instance, SLOs can postulate clauses like "95% of all requests in a day will be responded within 1 s, while less than 0.1% exceed 5 s", "The system allows up to 10,000 requests/s", or "The system will be unavailable for less than 3 h per year, with a maximum outage resolution time of 1 h". By consequence, continuous monitoring of all incoming requests has to be applied. As the system is looked at as a black box, the monitoring coverage can be limited to the externally visible and callable interfaces through which the client requests enter the system.

Detection of QoS problems: Predictive support for failure prevention is desired in order to improve a system's fault tolerance. If alarming system events or states indicating future performance drops are recognized early enough, action can be taken to adapt the system assembly or deployment. Therefore, responsiveness and scalability of the system components have to be observed and evaluated continuously. In a distributed system, a central controller that integrates and analyzes monitoring data allows tracing requests along all affected components to establish a system-wide

3. Self-Adaptive Performance Monitoring Approach

determination of cause-and-effect chains. If a performance degradation is indicated, adaptation decisions and actions concerning the monitoring coverage have to be derived such that the anomalous behavior can be reported and visualized. While for SLA compliance recording requests at system or component entry level is sufficient, now it becomes necessary to trace the component-internal control and data flow. The number and the choice of active probe measuring points should be adaptable at runtime. Filtering the set of active probes and measuring points on demand allows zooming into a component if it behaves unexpectedly. Zooming implies the activation of more (or less) measuring points in the application's control flow aiming at increasing (or decreasing) insight, e.g. into the operation call stack, effective loop iterations, or conditional branches taken. The activation control of probes and measuring points can either be applied manually or automatically. For self-adaptive control a set of guiding monitoring rules is required (see Sections 3.3 and 3.4.1).

Capacity planning support: To enable runtime capacity management, monitoring provides data needed for performance predictions and adaptation decisions. Capacity planning usually correlates application-level workload patterns with utilization data of software and hardware resources. Workload trends are based on previously recorded time series of component-level interaction. Resource utilization data is typically either gathered by the runtime container (see Section 2.3.3) or by the underlying operating system. Basic OS-level tools such as `iostat`, `vmstat`, or `netstat` are integrated to monitor CPU load, disc access, memory utilization, and network traffic. Based on the collected information about workload and utilization, a capacity management controller that aims at resource efficient operation can induce architectural reconfiguration at runtime. These reconfiguration operations are for example the allocation of further server nodes, or component replication or migration between the nodes being already available for deployment [van Hoorn et al., 2009a]. Corresponding to the impact severity of the adaptation actions to be made, it might be as well desired to change the granularity of decision-supporting monitoring data at runtime.

3.2. Self-Adaptive Monitoring Process and Architecture

Recognition of usage patterns: Interface designers and marketers are interested in how users actually interact with the system. For that purpose, monitoring can deliver relevant information, e.g. how frequently the provided services are called from different user groups, or how the users navigate through a graphical user interface (GUI) within a session. As the requirements concerning usage pattern recognition may change unexpectedly, it is appropriate that the monitoring coverage is adaptable without any need for rebuilding and redeploying the system components.

3.2 Self-Adaptive Monitoring Process and Architecture

This section introduces the proposed method for self-adaptive performance monitoring of component-based software systems, as well as the underlying monitoring architecture. The monitoring process consists of 7 activities described in the following. The sequence of these activities is illustrated by a partitioned activity diagram in Figure 3.1. The interrelation of the activities with the major architectural components of the monitoring framework is annotated to the component diagram shown in Figure 3.2. The approach utilizes and extends the Kieker framework, as presented in [Ehlers and Hasselbring, 2011b].

3.2.1 Probe Injection

As explicated in Sections 2.2.5 and 2.3, the system that should be monitored has to be instrumented with probes. A probe is positioned at measuring points at which observation of control or data flow is of interest. To measure response times, measuring points are placed at entry and exit join points of component-internal operations. These include the services provided by component interfaces, but may also cover encapsulated methods or code blocks.

3. Self-Adaptive Performance Monitoring Approach

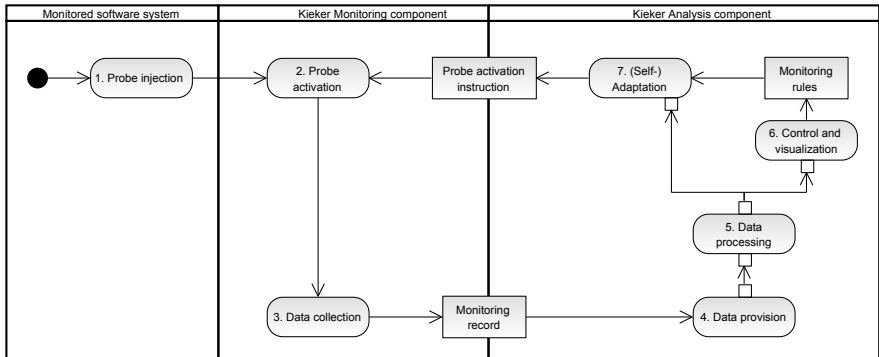


Figure 3.1. Adaptive monitoring process

As described in Section 2.3.2, AOP is an appropriate means to inject application-level probes into class methods in object-oriented software systems. Thus, Kieker’s default suggestion is to utilize the aspect weaving framework AspectJ for instrumentation. It has to be remarked that fine-grained instrumentation of method-internal code blocks is not possible with AspectJ. There exist several research approaches that extend the AspectJ join point model in order to enable interception and weaving of finer code blocks: e.g. Harbulot and Gurd [2006] provide a pointcut to select loops, Xi et al. [2009] introduce a pointcut for synchronized blocks, and Akai and Chiba [2009] propose a generic pointcut to identify arbitrary code regions. In the following, it is assumed that the monitored system is either well-designed or fairly refactored, so that method bodies are of clearly comprehensible complexity and method-level instrumentation is generally sufficient. In other cases, it is recommended to follow the proposed injection refinements named above or BCI approaches such as Javassist. To impurify the code by placing the probes manually, remains the least valued choice for instrumentation.

In addition, middleware interception techniques such as JMX are recommended to collect utilization data for active hardware resources and passive software resources (see Section 4.3.4).

3.2. Self-Adaptive Monitoring Process and Architecture

3.2.2 Probe Activation

As illustrated in Figure 3.2, Monitoring Probes are triggered from various measuring points embedded in the monitored software system. The probes are part of an instance of the Kieker Monitoring component, being called “*monitoring agent*” in the following. Regarding a distributed software system, a monitoring agent is deployed on each execution container and interconnected with the hosted components of the monitored system, e.g. by load-time aspect weaving. Each monitoring agent is supervised by a single Monitoring Controller. The controller manages which probes and measuring points are currently activated. Further, it provides access to a persistence unit for logging the monitored data (see Section 3.2.3) and provides an Adaptation Interface to be reconfigured externally (see Section 3.2.7).

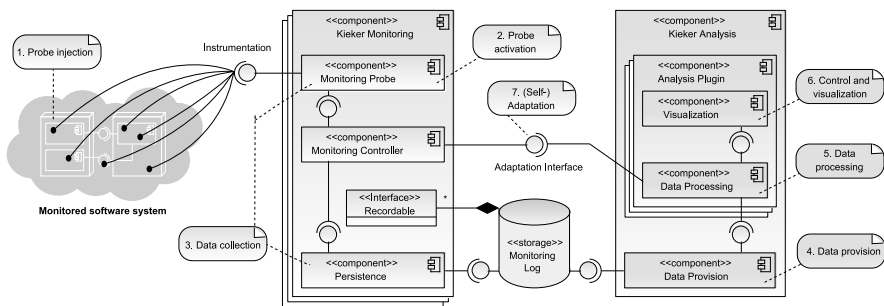


Figure 3.2. Adaptive monitoring architecture, cf. [Ehlers and Hasselbring, 2011b]

Considering a software system in productive operation, it is not feasible to record each observed occurrence of any probe at any measuring point. In fact, this restriction is caused more by the great number of expected requests/s than by the injected number of measuring points. Thus, an initial monitoring rule set has to be configured prescribing which probes are where and when active. The monitoring rules can be maintained and changed during runtime (see Sections 3.2.7 and 4.4). The possibility to enable and disable probes incorporates the requirement of the monitoring granularity to be adaptable at runtime. The selective activation of a measuring point may

3. Self-Adaptive Performance Monitoring Approach

depend on its call stack level, the responsiveness of the related operation, the current workload, a random probability, etc. The periodic activation and deactivation of measuring points can be understood as a workaround for runtime AOP or a subset of *partial behavioral reflection* [Tanter et al., 2003].

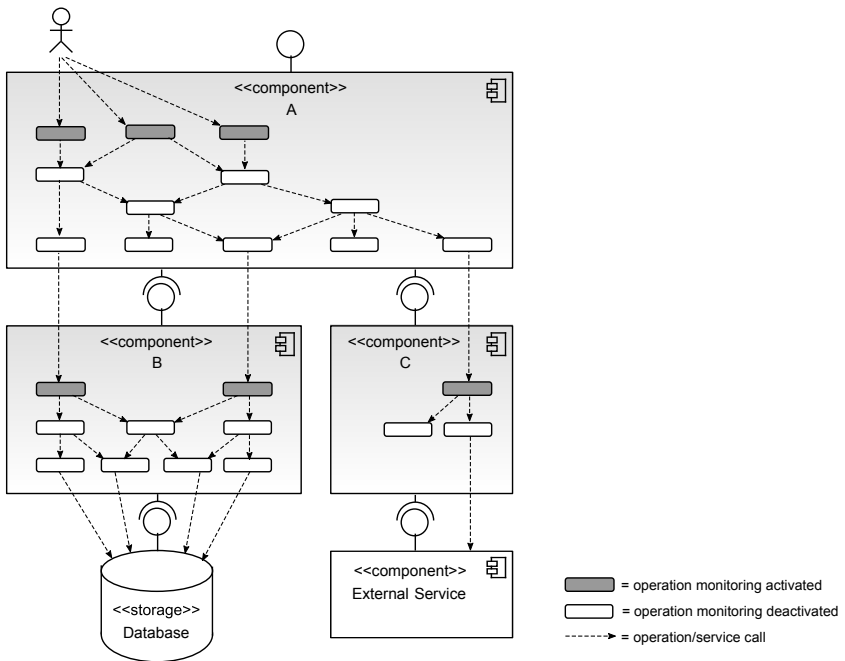


Figure 3.3. Software system with operation intercepting probes

Figure 3.3 depicts a sample software system made up of three components, each with several internal operations. For each operation, monitoring is either activated or deactivated. In the displayed system state, only operations serving as part of a component's provided interface are activated. Monitoring of all component-internal operations is deactivated. This is an appropriate initial configuration for the monitoring rules. At runtime, the monitoring coverage can be intensified to inspect the interior control flow of a component in case it does not behave as expected.

3.2. Self-Adaptive Monitoring Process and Architecture

3.2.3 Data Collection

Kieker comes along with a set of different aspects that allow the recognition of service call entries incoming via different interface technologies, including HTTP Servlets, JAX-WS, JAX-RS, EJB 3.1, or JMS. These aspects intercept specific framework methods, e.g. `javax.servlet.http.HttpServlet.do*(..)` for any HTTP request processed by a Servlet or `@javax.jws.WebMethod *(..)` for JAX-WS annotated interface methods. Further, Kieker provides an aspect to intercept application-specific operations. Typically, this includes all methods being part of a component's realization that contributes to the system's application logic and that is in the scope of monitoring. To define the monitoring scope for Java applications, practical solutions are to confine it on a set of package patterns, e.g. `com.example.*`, or to introduce monitoring-specific annotations.

Each request is tagged with an id when it enters the system. By means of the framework aspects, it is possible to trace a request over any inter-component and inter-server communication protocol throughout the system. To track a relation between a caller and its callee, the probe intercepting the calling operation can either store information about the measuring point in thread local memory or attach it to the meta-data of the service call. The latter is appropriate, if the call is asynchronous or overlapping container borders, i.e. multiple threads are involved. The probe linked to the called operation receives the information about its calling context and saves it in a monitoring record. Later, analyses will process the recorded calling context information of all operation calls related to a single request and reconstruct a monitored request trace (see Section 3.2.5). Besides calling context information, probes gather performance related metrics such as response times, call frequencies (to determine throughput), or resource utilization.

As monitoring data is collected at different nodes of a distributed system, it has to be carried together for system-wide analysis. Figure 3.2 shows that each monitoring component pushes its records into a central repository called Monitoring Log. Recording might be applied to any persistent storage repository like a file system, a database, or a buffered message queue.

3. Self-Adaptive Performance Monitoring Approach

Therefore, Kieker provides different persistence components to support several of these storage alternatives. As indicated in Figure 3.2, probes can collect arbitrary data records and push them into the log, on condition that the record class implements a generic Kieker interface for Recordable entities. It has to be noted that the permanent writing operations provoke a major part of the monitoring cost (see Section 5.3.1).

3.2.4 Data Provision

While the Kieker Monitoring component is deployed several times, with one instance at each execution container, it is sufficient to run only one instance of the Kieker Analysis component. The Kieker Analysis component frames a client application addressing the system's performance engineer. Its implementation, which is presented in Chapter 4, is realized as a tool based on Eclipse RCP and EMF. Basically, the data stream in the Kieker Analysis component follows the pipes-and-filters pattern [Taylor et al., 2009]. The monitored records serve as input for the piped analysis data stream. The first and lowest-level filter is a Data Provision component, connected to a dedicated monitoring agent. A monitoring agent provides information to such a data provider about its Monitoring Log used for record transfer. Due to this information, a data provider can connect to the log and continuously receive the incoming monitoring records and forward them to subsequent analysis filters.

3.2.5 Data Processing

The proceeding filters of the analysis data stream are joined together via a plugin mechanism. The analysis component is constructed to be easily extensible with (third-party) Analysis Plugins containing Data Processing and/or Visualization components, as indicated in Figure 3.2. Such data processing or visualization filters can be subscribed to other filters supplying data, provided that the filters' input and output ports match. Visualizations

3.2. Self-Adaptive Monitoring Process and Architecture

that display the analysis results are usually a sink of the data stream. Each filter can implement any kind of processing based on the records received from its predecessors. In case of a distributed system, a filter that is subscribed to multiple log-reading data providers serves as a consolidation filter that merges records from different monitoring agents. The incoming records deliver pieces of information that can be assembled or visualized in different ways. Records are related to tracing, responsiveness, and utilization as follows.

Tracing records: Tracing enables to recap how one or more client requests have been executed by the software system under inspection. Tracing can be applied at different compositional abstraction levels, e.g. studying interaction at the level of components, classes, or methods. In any case, information about the calling dependencies between the interacting entities is collected. This information can be represented in different ways, e.g. by a dynamic call tree [Jerding et al., 1997], by a call graph [Graham et al., 1982], or by a calling context tree [Ammons et al., 1997]. The representations differ in their grade of accuracy and efficiency, and hence are suitable for different tasks. Request-individual call trees are the most accurate representation, but at the same time the most storage space extensive. In contrast, a call graph is less accurate than the other representations, and in favor the most space efficient.

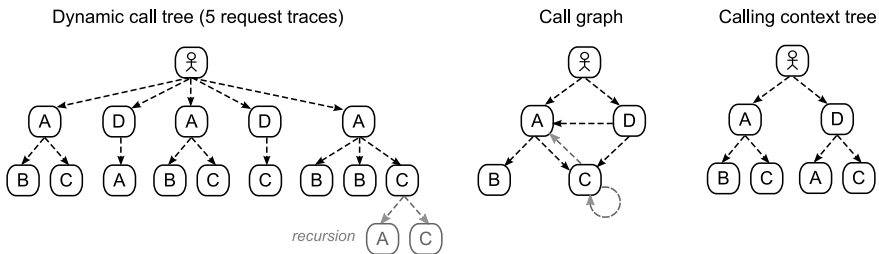


Figure 3.4. Dynamic call tree with corresponding call graph and calling context tree

A call trace is a set of all caller-callee interactions being observed while a system request is processed. A dynamic call tree captures all observed

3. Self-Adaptive Performance Monitoring Approach

request traces, as shown in Figure 3.4. Each path outgoing from the tree root indicates a single request trace. A dynamic call tree keeps track of the chronological sequence of all recorded, request-inherent interactions, e.g. operation invocations and returns. This includes information about the stack context in which any call was made. Loops and recursive calls are not pruned. A relevant demand to multi-user systems is that the call traces of simultaneous requests being concurrently serviced have to be isolated from each other. Regarding continuous monitoring, extracting and preserving all call traces is usually not feasible, since their depth as well as their (unique) number is unbounded. By consequence, some kind of aggregation is required.

A call graph is a tremendously more compact tracing representation. As in a dynamic call tree, the nodes of a call graph are the interacting entities, e.g. components. However, each entity is represented by a unique node and occurs only once in the graph (see Figure 3.4). A directed edge between two nodes indicates a possible call from the outbound node to the inbound node in the direction of the pointer. The maximal size of a call graph is bounded by the number of observed entities. To achieve compactness, possibly relevant information about the sequencing and the stack context of calls is dropped. Other than a dynamic call tree, a call graph is said to be context-insensitive. It does not preserve the calling context, i.e. the circumstances under which a call was made.

Unlike the other tracing representations, the extraction of a call graph is not limited to runtime monitoring but can be employed as well through means of static and offline reverse engineering [Trofin and Murphy, 2008; Sundaresan et al., 2000]. With static call graph extraction, the set of possible interactions is certainly discovered in completeness, while dynamic approaches will rather detect the interactions that are effectively executed at runtime. A comparison of both can help to find cases that are never or rarely taken, e.g. under exceptional input parametrization. The major disadvantage of a static technique is that the call graph cannot be augmented with additional performance information of the running system.

3.2. Self-Adaptive Monitoring Process and Architecture

A calling context tree (CCT) is an intermediate representation, which is more accurate than a call graph and less extensive than storing every single call trace. All requests with the same trace are consolidated in a CCT. Though the metrics of identical traces are merged, the stack context of each call is preserved (see Figure 3.4). The breadth of a CCT is limited by the number of observed entities. To handle recursive calls, the depth of a CCT is restrained artificially by defining that a node is equivalent to an ancestor node that represents the same entity [Ammons et al., 1997].

Rohr et al. [2008] define three classes of calling context equivalence for operations and their observed executions. In the following enumeration, operations are termed more universal as arbitrary caller and callee sites, and operation executions are generalized to directed calls from one site to another.

- *Caller-context equivalence*: Two calls to the same callee site are caller-context equivalent, if they are called from the same caller site.
- *Stack-context equivalence*: Two calls to the same callee site are stack-context equivalent, if the paths in the corresponding traces from the callee to the root are equal.
- *Trace-context equivalence*: Two calls to the same callee site are trace-context equivalent, if the corresponding traces are equal and the callee is at the same position in the trace.

Regarding these equivalence classes, a CCT preserves the stack context, but not the trace context as dynamic call trees do. A call graph only preserves the caller context, if context-related metrics are saved at the ingoing edges of a node (as these might be more than one) and not at the nodes themselves. A simplified class model for a CCT is depicted in Figure 3.6.

Responsiveness records: A response time record contains information about the start and end time of a single operation execution. It does not matter whether the operation is a high-level component service or a low-level class method. Moreover, the record tuple references one or more compositional entities that embed the operation, e.g. a component, a class, or both. As stated above, the record can also point to a caller, a call stack, or

3. Self-Adaptive Performance Monitoring Approach

a whole trace the operation execution is related to. For analysis purposes, the performance engineer is interested in an aggregated view over multiple operation executions.

According to the multidimensional data model [Agrawal et al., 1997], the record values can be distinguished between quantitative measures that are the objects of analysis and qualifying dimensional values. Each measure depends on a set of dimensions, which provide the semantic context for the measure [Kimball and Ross, 2002]. To illustrate the multidimensional view on a measure, the metaphor of a data cube is widely used, with each dimension being an axis of the cube. The length of a dimension is determined by the number of its members. Each member represents a distinct observed dimension value. Among the dimension members, there can exist hierarchical relationships across several levels. The viewpoint on the measures can be changed by zooming in or out along the aggregation paths of a dimension hierarchy, by fading in or out some of the dimensions, or by filtering a subset of dimensions members. The terms of the multidimensional data model are illustrated in Figure 3.5.

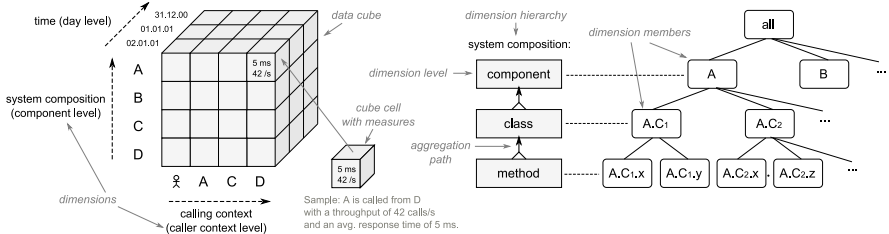


Figure 3.5. Cube and dimensions in a multidimensional data set

In case of responsiveness records, the measures being of primary interest are response times and throughputs. Response time is computed as the duration between an execution's start and end time. Appropriate units are milli- or nanoseconds. Throughput can be captured by implementing a frequency counter that is related to time, e.g. calls/s. Another interesting measure often is the transferred amount of data, e.g. MB/s. A common dimension of all measures is time. Time has evident hierarchy levels like

3.2. Self-Adaptive Monitoring Process and Architecture

month, day, hour, minute with e.g. “Dec. 2000” and “Jan. 2001” being members of the month level and “31.12.00” being a member of the day level.

As indicated in Figure 3.5, further dimensions are the degree of system composition and the calling context. In case of component-based object-oriented software, the composition level ranges from components over classes to methods. The levels of the calling context dimension aggregate measures with a common calling context either related to a caller, a stack, or a trace (according to the equivalence classes explicated above). Figure 3.5 shows a multidimensional viewpoint on the measures response time and throughput from the following dimension levels: system composition level “component”, calling context level “caller context”, and time level “day”.

Another dimension is the type of data acquisition. Its members are “predicted” indicating that the measures are won through model-based simulation, and “measured” indicating that the measures have been monitored from a running system. The members of a lower data acquisition level could name specific tools or probes that collected the measures.

A secondary measure is the response time that is expected a priori of an operation execution. The expected response time is not measured but derived from a time series of previous executions of the same operation in the same or a similar context. Based on the deviation of observed and expected response time, a further derived measure is the anomaly score assigned to each operation execution (see Section 3.4.3).

Utilization records: The utilization measures are recorded with reference to a software or hardware resource with a dedicated capacity. For analysis purposes, it can be interesting to add a workload dimension in addition to the capacity dimension. The members of the workload dimension specify load rates of incoming user requests. It has to be reminded that each utilization value is a sample snapshot standing for a time interval. The probes gain the utilization data by means of OS-level tools and aggregate it

3. Self-Adaptive Performance Monitoring Approach

by averaging. Analysis plugins visualize resource utilization curves related to time- or workload axes.

3.2.6 Control and Visualization

It is a major requirement of a monitoring tool to provide a control panel that conveniently visualizes the monitored data for analysts. The Kieker Analysis component, as presented in Chapter 4, serves as such a control panel and comes along with a set of plugins that implement the graph representations explicated above to picture tracing and compositional dependencies. Section 4.3 contains example visualizations providing an impression of the Kieker control panel. For instance, CCTs or call graphs of either components, classes, or methods can be constructed to visualize viewpoints of different calling context and system composition levels (see Section 4.3.6). Other plugins offer time series curves capturing responsiveness and resource utilization over time (Section 4.3.4), UML sequence diagrams (Section 4.3.2), use case transition graphs based on session-internal navigation patterns (Section 4.3.3), or online analytical processing (OLAP) features (Section 4.3.5). Different third-party libraries such as Graphviz², Eclipse Zest³, or JFreeChart are used for graph and chart visualization. As stated, the Kieker Analysis component can easily be extended with further plugins if required.

3.2.7 (Self-)Adaptation

Concentrating on the adaptiveness of monitoring probe activation, the MonitoringAdaptation plugin described in detail in Section 4.4 is of particular interest. It is designed to be one of several plugins integrated into the Kieker Analysis component and enables to (re)configure the set of active probes and measuring points having previously been injected into the system.

² <http://www.graphviz.org/>

³ <http://www.eclipse.org/gef/zest/>

3.2. Self-Adaptive Monitoring Process and Architecture

For each instrumented monitored agent in a distributed system, a probe can either be activated or deactivated. For example, an operation interception probe can be activated at agent *A* while a resource utilization probe is deactivated. At the same time, the activation state of both probes can be vice versa at agent *B* (see Figure 4.16).

Each probe has a set of measuring points from which it can be triggered. Not every possible measuring point of a probe is desired to be permanently activated. Thus, the `MonitoringAdaptation` plugin allows a performance engineer to dispose monitoring rules, each one being related to one or more probes. A monitoring rule is expressed by an Object Constraint Language (OCL) expression [OMG, 2010]. It specifies a set of measuring points in form of method signatures at which the referenced probes should be activated. The evaluation and appliance of the monitoring rules can either take place manually, i.e. released on a click in the control panel, or automatically, i.e. repetitive each time a specified time period has elapsed. As shown in Figure 3.2, each monitoring agent provides an `Adaptation Interface` via its controller. The `MonitoringAdaptation` plugin provides an extended `Data Processing filter` (see Section 3.2.5) that allows starting a concurrent thread for continuous evaluation of the monitoring rules. When a violation of a rule premise is detected, the `MonitoringAdaptation filter` can execute the adaptation autonomously (see Section 3.4.1). Goal-oriented self-adaptation is based on the possibility to refer to (performance) attributes in the monitoring rules that change their values during runtime, e.g. responsiveness metrics and derived anomaly scores (see Section 3.4.3). The continuous and autonomous evaluation and appliance of the monitoring rules implements the adaptation planning and execution activities of the generic control feedback cycle is described in Section 2.4.2. The following section provides more details on how the monitoring rules can be expressed by means of OCL.

3. Self-Adaptive Performance Monitoring Approach

3.3 Expressing Monitoring Rules

The proposed adaptation process works like a simple rule-based expert system that makes use of deductive reasoning to reach a decision [Hayes-Roth, 1985]. The performance engineer acts as the expert who formulates the rules of inference. A rule of inference is a function that takes one or more premises and returns a conclusion. The premises specify preconditions, which if they evaluate to true imply the conclusion. Forward chaining is to derive conclusions given a base of rules and preconditions. The term forward chaining is commonly used in the domain of artificial intelligence. Its origin in logical reasoning is the argument form *modus ponens* denoted in sequent notation as $A \rightarrow B, A \vdash B$, i.e. "If A , then B . A . Therefore, B ." The argument form has two premises. The first premise is the if-then-rule itself. The second premise is that A evaluates to true. Given the rule $A \rightarrow B$ and the fulfilled precondition A , consequently B can be concluded.

To adjust the monitoring coverage, the desired conclusions are either to activate a set of currently disabled probe measuring points, or to passivate a set of currently enabled probe measuring points. Other conclusions are generally conceivable, but are not implemented (see Section 4.4). The set of measuring points to change activation for is addressed in the rule's premise. A monitoring rule's premise consists of an OCL context element and an OCL expression. The context element specifies the context in which the expression will be evaluated. Typically, the adaptation filter itself is an appropriate context element, because it allows navigation to all records that have been supplied by preceding data processing filters.

In the following, example monitoring rules are discussed. The first one starts with a minimal set of active measuring points, comprising only class methods (as concrete compositional entities) being positioned at the topmost level of a CCT. It is assumed that the context element referenced by the OCL identifier *self* is set to a simplified CCT instance (see Figure 3.6) having previously been delivered to the adaptation filter.

3.3. Expressing Monitoring Rules

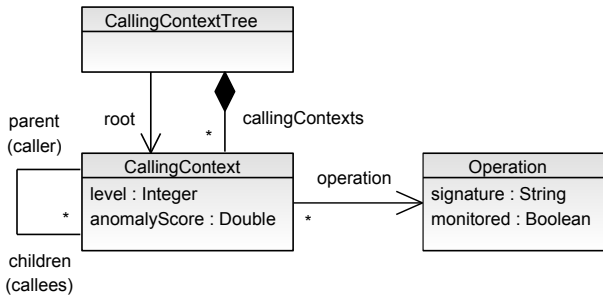


Figure 3.6. Class model of a calling context tree

Monitoring rule R_1 : If an operation is at the top of a traced call stack (level 1 of the CCT), then enable the probe measuring points required to intercept and monitor invocations of this operation. The corresponding OCL expression is:

```
self.callingContexts→select(level = 1)→collect(operation)
```

The second example monitoring rule is somewhat more complex. It aims at intensifying the monitoring coverage within a component if it behaves anomalous.

Monitoring rule R_2 : If the precondition of R_1 is fulfilled or if the corresponding caller operation is already monitored and behaves anomalous, i.e. its anomaly score exceeds a specified threshold t , then enable the probe measuring points required to intercept and monitor calls to the callee operation. OCL expression:

```
self.callingContexts→select(level = 1 or
(parent.operation.monitored and parent.anomalyScore > t))→collect(operation)
```

OCL provides several built-in collection operations to enable powerful ways of projecting new collections from existing ones [OMG, 2010]. The expressions above make use of the operations select and collect. In case

3. Self-Adaptive Performance Monitoring Approach

of the monitoring rules, the demand is to reduce the set of all measuring points to a distinct selection. The operation `select` satisfies this demand, as it selects a subset of a collection based on a boolean expression. The OCL simple syntax form is `collection→select(boolean-expression)`. It can be compared to the selection operation σ as its counterpart in relational algebra. The operation `collect` serves to come up to a derived collection that contains different object types than the collection it is originated from, i.e. the new collection is not a subset of the original one. The OCL simple syntax form is `collection→collect(expression)`. Its counterpart in relational algebra is the projection operation π . A sample scenario making use of the rule R_2 is discussed in Section 3.4.1.

Apparently, there exist several alternatives to OCL to describe rules by means of a descriptive language such as XQuery [W3C, 2010], SQL [ISO/IEC, 2008], or Alloy [Jackson, 2002]. Like OCL, the two first ones fulfill the criteria to be widely accepted in practice and to be specified by a committee of researchers and industry-leading companies. The major benefit of OCL is that it primarily addresses object-oriented runtime models instead of XML trees like XQuery or relational schemas like SQL. OCL allows for navigation through any model or meta-model that is based on MOF [OMG, 2011d]. OCL is a core part of the MOF specification QVT (Query/View/Transformation) that provides standards for model to model transformations [OMG, 2011c]. The latest MOF specification, particularly its lightweight core called Essential MOF (EMOF), has substantially been influenced by the hands-on development experience from the EMF project. EMOF closely resembles EMF's Ecore meta-model being the basis for the specification of arbitrary (domain-specific) modeling languages. Both share the ability to specify classes with structural and behavioral features, inheritance, packages, and reflection. Because of their similarity, EMF supports EMOF as an alternate XMI (XML Metadata Interchange) serialization of Ecore [Steinberg et al., 2008].

OCL is mostly known for its purposes to specify invariants on classes, pre- and postconditions on operations, or guards annotated to UML diagrams. Despite that, the first purpose named in the OCL specification suggests

3.4. Performance Anomaly Detection

OCL to serve as a query language. Indeed, the monitoring rule premises can be seen as queries for a set of measuring points to be (de)activated. The Kieker MonitoringAdaptation plugin (see Section 4.4) makes use of the EMF sub-project Query2⁴, which allows constructing and running queries on EMF models by means of OCL.

A further alternative to OCL is the creation of an individual domain-specific language (DSL) using a tool such as Xtext⁵ [Efftginge and Voelter, 2006]. The advantage of such a DSL is that it can remove the complexity of OCL, as being tailored to the specific requirements of monitoring rules. On the other hand, a DSL would come along with the typical disadvantage of an isolated application having to be explicitly learned with any chance of reuse. As this con is rated more severe, the decision turns out for OCL.

3.4 Performance Anomaly Detection

This section focuses on the goal to detect the origins of QoS problems or other system failures that effect a change in the system's normal behavior as perceived by its users. System failures can be caused by faults at any virtualization layer, cf. Figure 2.20 [Zeller, 2009]. Particularly, severe failures that decrease the overall availability of the delivered service have to be repaired as quickly as possible. A major part of the failure recovery time is required to localize a failure's root cause [Salfner et al., 2010]. Kiciman and Fox [2005] refer to an estimation that 75% of the recovery time is spent just for fault detection. They surveyed that it often takes days to search for a failure's cause, while it can be fixed quickly once the fault is found.

⁴ <http://www.eclipse.org/modeling/emf/?project=query2>

⁵ <http://www.eclipse.org/Xtext/>

3. Self-Adaptive Performance Monitoring Approach

3.4.1 Continuous Evaluation of Monitoring Rules

In case that harmful faults induce anomalous runtime behavior at application level, the approached solution allows adapting the monitoring coverage on demand. If the adaptation is conducted manually, the human decision to change the set of active probes or measuring points is usually based on a previous interpretation of performance visualizations provided by other plugins. A typical situation is that time series curves indicate a decline of a service's throughput though the workload did not increase. A performance engineer who recognizes this incidence is interested in the cause of the phenomenon and therefore tries to activate more measuring points in the affected components. Afterwards, it takes a while until enough analysis-relevant data has been collected via the newly activated probe measuring points. The approached self-adaptive monitoring process aims at reducing this potentially business-critical wait time that delays a failure or anomaly diagnosis.

The continuous and autonomous evaluation of the monitoring rules is part of the monitoring system's control feedback cycle (see Figure 2.26). It implements the activities of adaptation planning and -execution. The preceding monitoring and analysis activities that provide the decision-relevant input data have to be conducted in advance.

- *Monitoring*: In a distributed system, monitoring takes place in a decentralized manner. Each node of the system is instrumented with a monitoring agent, which collects monitoring data via probes and delivers the data to a central repository (see Section 3.2.3).
- *Analysis*: The data is consolidated and analyzed in a single analysis component instance. It provides a graphical control panel that allows configuring analysis-oriented data streams through different data processing filters (see Sections 3.2.4, 3.2.5).
- *Adaptation planning*: An adaptation filter can be placed at the end of such a data stream. It supports to express monitoring rules to be checked continuously. The rules are formulated based on the data model of the preceding input filters (see Section 3.3).

3.4. Performance Anomaly Detection

- *Adaptation execution*: The evaluation of the monitoring rules may induce the activation or deactivation of probe measuring points. By consequence, the probes will observe different aspects of the system's runtime behavior.

In the following sample scenario, the monitoring rule R_2 , which has been introduced in the previous section and allows the detection of operations with anomalous responsiveness, is applied. Figure 3.7 shows three succeeding system states with different sets of activated measuring points. The change of the monitoring coverage from one state to the next is achieved automatically by means of self-adaptation.

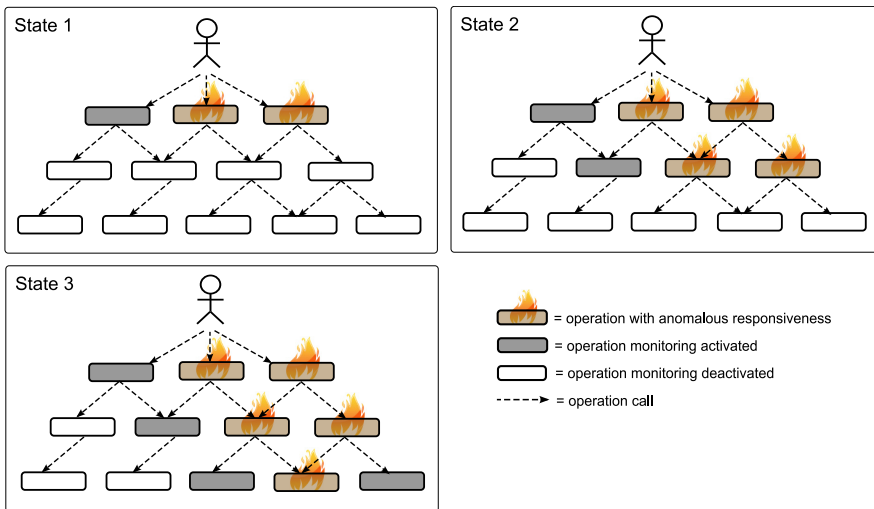


Figure 3.7. Succeeding states of monitoring coverage for anomaly detection

In *State 1*, only operations at the first level of the call stack are observed. As indicated by a flame icon, it is discovered that some of these operations behave significantly unexpectedly. The concrete anomaly detection method made use of is described in Section 3.4.3. As shown in *State 2*, the evaluation and appliance of the rule R_2 leads to the measuring point activation for all callees of the operations indicated as anomalous. In the sample scenario, it

3. Self-Adaptive Performance Monitoring Approach

turns out that two of three newly observed operations at the second CCT level are anomalous as well. Consequently in *State 3*, the next adaptation iteration provokes the measuring point activation of their callees. Finally, Figure 3.7 demonstrates that only one exceptional operation at the third CCT level is the root cause of all higher-level anomalies. It should be plausible that a manual exploration of such an exemplary cause-and-effect chain is much more time-consuming and error-prone than an automated proceeding. A contribution of the self-adaptive monitoring approach is to save this time and effort.

By default, every probe measuring point is activated initially. When a measuring point is triggered for the first time, its existence becomes known to the central analysis instance. Afterwards it can immediately be deactivated. The time interval between the two succeeding evaluations of the monitoring rules has to be set up. It has to be long enough to measure reliable response times before a measuring point is deactivated again, but as well short enough to react flexibly and without distracting delays. A value in the scale of a couple of seconds is appropriate, if immediate deactivation of measuring points having just been activated is prevented. To exclude short-term oscillation of the monitoring coverage, an additional *activation freeze time* parameter is introduced (see Section 5.4 for evaluation results). The implementation of the adaptation interface is explicated in Section 4.4.

3.4.2 Anomaly Detection Challenges

Anomalies are patterns in the monitored data that do not conform to the expected behavior. Finding these nonconforming patterns, also called outliers, is referred to as anomaly detection. Besides fault detection in software systems, anomaly detection is extensively applied in domains such as fraud detection for insurances and credit cards, or intrusion detection for secured (cyber)spaces [Patcha and Park, 2007].

As explicated in Section 3.2.5, responsiveness measures are collected in different dimensional contexts, e.g. operation response times related to

3.4. Performance Anomaly Detection

stack contexts. Therefore, an observed record consists of multiple attributes (multivariate) containing measures or contextual values. Whether the value of a measure is rated anomalous or not, strongly depends on the associated contextual values. Generally, anomalies can be classified into simple point anomalies, contextual anomalies, and collective anomalies [Chandola et al., 2009].

- *Point anomaly*: If a single record can be considered as anomalous regarding all the rest of the data (unrelated to any specific context values), then the record is termed a point anomaly.
- *Contextual anomaly*: If a record is anomalous in a specific context, but not otherwise, then it is termed a contextual anomaly. In any time series data, time is a native context dimension that assigns a position to each record in the total data set (see left curve in Figure 3.8).
- *Collective anomaly*: If a collection of records is anomalous regarding a related total data set (which may have been filtered to a specific context previously), then it is termed a collective anomaly (see right curve in Figure 3.8). The decisive criterion is that the single records being part of a collective anomaly would be classified as normal by themselves, but their occurrence together in a collection makes them anomalous.

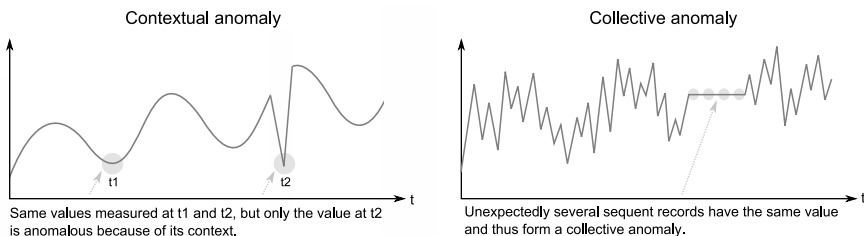


Figure 3.8. Contextual and collective anomalies

Regarding response time curves, contextual and collective anomalies have to be looked for. Contextual attributes such as the input parametrization or the stack context of an operation usually impact its response time significantly. Aside from, it provides more confidence not to signal a false alarm if

3. Self-Adaptive Performance Monitoring Approach

a collection of sequent outliers has been observed. A single operation execution whose response time does not conform to the expected value is not necessarily an alarm-worthy anomaly. Thus, a straightforward anomaly detection technique, which is to define a region confining normal behavior and declare any observation outside this region to be an anomaly, has to cope with some challenges:

Noise: Often observation data contains noise that is difficult to separate from anomalies. In contrast to anomalies, noise is a phenomenon in data that is not of interest for analysis, but complicates it. Noise can either be removed before the data analysis is performed, or it can be accommodated by immunizing the applied pattern recognition model.

Novel patterns: Anomaly detection is similar to the detection of novel patterns that occur for the first time in an observed time series. The difference between anomalies and novelties is that the latter are typically integrated into the normal behavior after being observed. The reason is that normal behavior evolves. Thus, a trained snapshot of normal behavior might not be representative in future. Newly emergent patterns such as trends and seasonal components have to be detected and updated continuously.

Domain specificity: What in fact makes out an anomaly depends on the concrete application domain. For example, small fluctuations concerning a software service's responsiveness could be tolerated as normal, while they would be a distressing anomaly in other domains, e.g. in medicine when measuring the human body temperature. Besides, it is generally difficult to define a normal region as the boundary between normal and anomalous behavior in most cases is rather fuzzy than precise.

Labeled data: Whether or not labeled data is available to train and to validate the model used for pattern recognition, restrains the selection of an appropriate technique. Labeled data that contains comprehensive and correct assignments between input and output values is hardly to obtain. As labeling is often done manually by human experts, it may be prohibitively expensive. Labeled data is a precondition for supervised anomaly detection,

3.4. Performance Anomaly Detection

e.g. to learn a classifier in a training phase. The training phase precedes the actual test phase during which the classifier is employed to classify incoming records. The faster the normal behavior changes, the more challenging is it to keep a classifier updated. Without labeled data, anomaly detection is limited to unsupervised techniques based on the implicit assumption that normal data instances are far more frequent than anomalies.

The choice of an appropriate anomaly detection technique depends on the above criteria as well as on the nature of input and output data. Each attribute of a data record may have a specific data type and range such as binary, categorical, or continuous. A variety of anomaly detection techniques (classification by nearest-neighbors, decision trees, Bayesian networks, backpropagation, support vector machines, clustering algorithms) can be adopted from recent data mining literature [Han et al., 2011; Nisbet et al., 2009] and pattern recognition literature [Bishop, 2006; Theodoridis and Koutroumbas, 2008].

3.4.3 Rating of Anomalous Responsiveness

In the context of application-level failure detection, any technique employed for the determination of anomalies has to be evaluated according to (1) its rate of failures that could not be detected (false negatives, type II error), and (2) its susceptibility to false alarm (false positives, type I error). As there is a trade-off between both, balanced thresholds that define the boundary between normal and anomalous have to be found. Particularly, a multitude of false positives reduces the efficacy of an automated approach. They cause unnecessary effort and emotionally blunt an analyst, who has to investigate and discard each false positive.

Figure 3.9 illustrates the possible wrong decisions in statistical hypothesis testing. Accordingly, anomaly scoring approaches are based on statistical inference, i.e. making decisions about a population based on the information contained in a sample from that population. The sample are the operation response times measured in the course of the monitoring process.

3. Self-Adaptive Performance Monitoring Approach

		Actual finding	
		Anomalous	Normal
Anomaly detection	Rated anomalous	True positive	False positive (type I error)
	Rated normal	False negative (type II error)	True negative

Figure 3.9. Anomaly detection errors: false positives and false negatives

Making use of the strategy design pattern, the Kieker Analysis component allows implementing and leveraging different algorithms to calculate anomaly scores. In the following subsections, two favored anomaly scoring approaches are described, which are implemented and evaluated in the later course of the thesis. The approaches are based on time series analysis (Section 3.4.3) and on distribution clustering (Section 3.4.3) respectively.

Time Series-Based Anomaly Scoring

This anomaly scoring procedure consists of the following four steps, which are described in detail below, after a brief introduction of response time determinants and statistical foundations.

1. Forecast expected response times for each operation (in dependence of the stack context) based on historical observations.
2. Test if a sample of newly observed operation response times has to be rated as normal or anomalous related to the expected value from 1.
3. Calculate an anomaly score for each operation based on the sequent rating of response times samples in 2.
4. Aggregate and correlate anomaly scores from 3. to higher levels of abstraction, e.g. component-level anomaly scores.

A software service has an expected response time that usually depends on the input parametrization and the current system workload. Known variations of parametrization and workload are reflected in the observed

3.4. Performance Anomaly Detection

series of response times. The historical data will be used for response time forecasting. If the measured response times will significantly deviate from the forecast values, an anomaly is reported. An anomaly may be caused by an unexpected variation of the service's request parametrization or of the service's number of concurrent users. A further cause can be a defect in the service's implementation (including defects in external services that the requested service depends on). Analyzing the monitoring data allows the exclusion of some possible causes, e.g. by observing that the hardware resources are not fully utilized and the system load is still in the expected range.

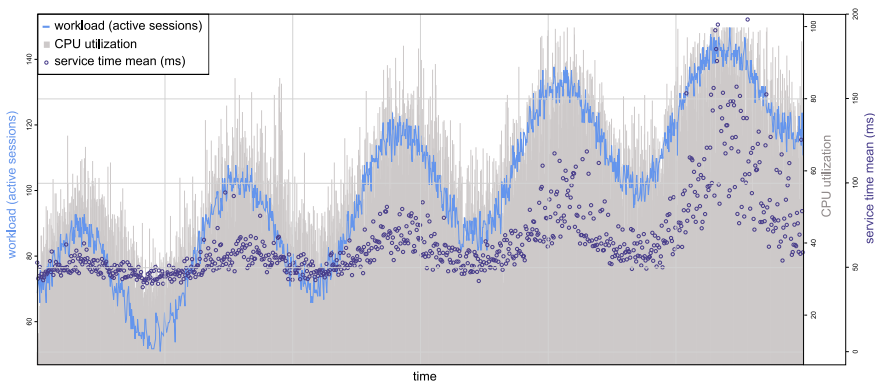


Figure 3.10. Time series of workload, CPU utilization, and response time mean

Figure 3.10 shows a software service's workload curve with a positive linear trend and a daily seasonal variation. The trend is caused by the incident that the number of users requesting the service increases continuously. The cause of the seasonal variation is that the service is used more frequently at daytime than at nighttime. In the shown example, the CPU utilization positively correlates with the workload indicated by the number of active users (Spearman rank correlation coefficient $r_s = 91,9\%$, Pearson correlation coefficient $r = 91,7\%$). Additionally, the measured mean response times correlate with the load, even though less significantly ($r_s = 82,1\%$, $r = 59,7\%$). At the beginning of the time series, the response times react only

3. Self-Adaptive Performance Monitoring Approach

slightly on increasing load, as the underlying hardware resources are not utilized heavily. Towards the end of the time series, the CPU is fully utilized for a while. During this time, the number of response time outliers increases. The outliers are first anomaly indicators, as the system capacity limit is nearly reached and the CPU tends to become a bottleneck (cf. Figure 2.7).

Like in most statistics studies, it is impossible to observe the entire population as the number of future operation executions is not limited. Therefore, a sample of observations is surveyed, which represents a subset of the population. The population is regarded as conceptual and is modeled by an assumptive probability distribution. Decisions do not rely on the model, but on the observed sample data. For the statistical inference to be valid, the sample has to be representative of the population. Consequently, it is desirable that the selection of a sample is a random experiment. A random experiment is defined as an experiment that can result in different outcomes, even though it is repeated under identical conditions. The set of all possible outcomes is denoted as the sample space Ω . The sample space can be discrete, if it consists of a finite (or countable infinite) set of outcomes, or continuous, if it embraces an interval of real numbers. In case of measuring response times in nanoseconds, the sample space is continuous. Each event E as a subset of Ω can be assigned with a probability $P(E)$, which quantifies the likelihood that E occurs as an outcome of the random experiment. Thereby, the axioms $P(S) = 1$ and $0 \leq P(E) \leq 1$ are to be complied. Each observation of an operation execution is regarded as a random experiment. A random variable X assigns a real number to each outcome of the sample space. The probability distribution of a random variable X associates possible values or value intervals of X with probabilities. The specification of a probability distribution depends on whether X is discrete or continuous. In the relevant case that X is continuous, its distribution is described by a probability density function f with

$$f(x) \geq 0, \quad \int_{-\infty}^{\infty} f(x) dz = 1, \quad \int_a^b f(x) dx = P(a \leq X \leq b).$$

3.4. Performance Anomaly Detection

The distribution function $F(x) = P(X \leq x)$ cumulates the probability that the random variable X is less than or equal a distinct value x . There exist a number of different named distribution families that define a parameterizable distribution function such as the normal, the log-normal, or the gamma distributions. Typical parameters characterizing an instance of a distribution family are its mean μ quantifying the expected value $E(X)$ and its variance σ^2 as a measure of the distribution's variability $V(X) = E(X - \mu)^2$. Parameter values significantly influence the shape and scale of the distribution function. With respect to further foundations of probability theory and statistics, the reader is referred to introductory literature such as [Montgomery and Runger, 2006] or [Ross, 2009].

Considering software components, the actual distribution model of response times is a priori unknown. Though response times are drilled down to a fine-grained contextual level at which distinct methods and their stack contexts are differentiated, this is not necessarily sufficient to expect a deterministic response time. There can exist further contextual attributes that influence an operation's response time such as its input parametrization or the current system workload. To measure workload might be possible due to observing the concurrent threads and the CPUs' utilization while a service is conducted. These measures can be taken as workload indicators. However, to measure an operation's input parametrization in completeness is practically not feasible. Looking at generic component services, it may not only be the input arguments of the interface signatures that effect the response times, but also some global component-internal parameters. The component-internal state is often not transparent for a client initiating a service request. Nevertheless, it can have a decisive influence on the control and data flow, and on the resulting resource demand. Even, if one could assume that only the input arguments are effecting an operation's response time, it still remains expensive to serialize all input arguments, especially those that are of complex types. This leak to state explosion can be avoided by enforcing a performance engineer to mark the performance-effecting service parameters in a design-oriented performance model. For example, as it is done in the RDSEFFs of the Palladio Component Model (see Section 2.2.4). For runtime measurement purposes, it has to reminded

3. Self-Adaptive Performance Monitoring Approach

that the parameter values to be recorded should be quickly serializable. As often the performance-relevant parameters are unknown in advance or a design-oriented performance model does not yet exist, the anomaly scoring procedures presented here approach a different solution: It is assumed that there is no ability to separate all context-determinant impact factors. Thus even from a fine-grained contextual viewpoint, response times can be arbitrarily distributed and do not necessarily converge to a parametric distribution instance.

1. Response time forecast: Response time observations being ordered in time form a univariate time series x_1, x_2, \dots, x_T . A time series is regarded to be a realization of a stochastic process $\{X_t : t \in T\}$, which is a sequence of time-indexed random variables defined on some sample space Ω . A basic assumption in time series analysis is that some patterns observed in the past will remain in the future. Hence, time series analysis can be used for short-term forecast purposes based on historical data: $x_1, x_2, \dots, x_T \rightarrow \hat{x}_{T+1}, \hat{x}_{T+2}, \dots$ (observations \rightarrow predictions). Forecasting based on a time series abstracts away from any technical or economical interrelations. It is assumed that characteristic features of the stochastic process can be recovered from the historic data. Given that the stochastic process is unknown, an appropriate process model has to be found. The model should fit to the observed time series, as it is assumed that the time series has been generated from this process and future time series values will continue to arise from the same process. Different common models for stochastic processes (based on ARIMA forecast models, see below) are implemented and evaluated regarding their ability to predict the response times of software components (see Sections 4.3.6 and 5.1).

Time series analysis aspires to capture some kind of regularity that exists over time in the behavior of an observed time series. The basic idea of regularity is embodied in a concept called stationarity. To gain compactness, it is practically sufficient to characterize a stochastic process by its random variables' first and second moments. A stationary process implies that these statistical properties μ and γ_j will not change with time.

Stationary Time Series cf. [Shumway and Stoffer, 2006]

A stochastic process $\{X_t\}$ is (weakly) stationary,

if its mean function $\mu(t) = E(X_t)$ is constant and does not depend on time t , i.e. $\mu(t) = \mu$,

and if its (auto)covariance function $\gamma_j(t) = Cov(X_t, X_{t-j})$ does not depend on time t , but only on the lag j , i.e. $\gamma_j(t) = \gamma_j$.

To estimate the statistical properties from a single observed time series, the underlying process has to fulfill another assumption: Generally, estimators assume independent random variables. In case of time series, the subsequent realizations of the stochastic process typically depend on each other. In fact, the identification of the specific dependency structure is a major aim of the time series analysis. To attain consistent estimators for the statistical properties from longitudinal sample data such as response time sequences, the process has to be ergodic. A stationary process is ergodic if the covariances decrease with greater lags j , i.e. $\sum_{j=0}^{\infty} |\gamma_j| < \infty$. The observed response times in a software system fulfill this precondition.

Unfortunately, a time series of response times will not necessarily be stationary, particularly if different workload intensities are not separated from each other by means of a workload dimension. Typically, the system load has a direct impact on the responsiveness and load rates vary caused by trends or seasonal variations. The measured response times will clearly correlate with the workload, as shown in Figure 3.10. The time series can be thought of as being composed of multiple components, e.g. $X_t = T_t + S_t + I_t$ in an additive model where T_t is the trend component, S_t is the seasonal component, and I_t is the irregular noise component.

In case of non-stationary processes, the original process can be transformed to a stationary process by the prior appliance of filters. For instance, a non-stationary process with a linear trend is filtered by building pairwise differences $\Delta x_t = x_t - x_{t-1}$. The new time series made up of these first-order differences $\{\Delta x_t\}$ will be stationary. Differencing can easily be repeated

3. Self-Adaptive Performance Monitoring Approach

to higher orders, e.g. $\Delta^2 x_t = \Delta x_t - \Delta x_{t-1}$, to remove more complex trends. Seasonal variations of known period length may be quantified and adjusted by means of smoothing filters. To test whether a process is stationary or not, so called unit root tests can be applied [Wei, 2005].

To predict future response times, the approached anomaly scoring procedure allows choosing from one of multiple forecasting models. Each forecasting model assumes the time series to be generated from a different stochastic process. The implementation of the Kieker PerformanceAnomalyAnalysis plugin (Section 4.3.6) covers the forecasting models listed in the following. Concerning their empirical accuracy, they are compared to each other in Section 5.1.

In the following equations, \hat{x}_{t+1} is a forecast value for the future point in time $t + 1$, \hat{x}_t is a prior forecast value, and x_t is an actually observed value at prior time t . For further details concerning the listed forecast models refer to [Box et al., 2008; Wei, 2005].

Single exponential smoothing (SES):

$$\hat{x}_{t+1} = \beta x_t + (1 - \beta) \hat{x}_t$$

The SES model with the smoothing factor β ($0 < \beta < 1$) is a special case of the generic ARIMA (autoregressive integrated moving average) forecast model. More precisely, it is equivalent to the ARIMA(0,1,1) model with no constant term, i.e. $\Delta X_t = \theta \varepsilon_{t-1} + \varepsilon_t$ with $\theta = \beta - 1$. The process is made up of a single integration I(1) and a first-order moving average process MA(1). The white noise error represented by ε_t can be neglected in the forecast equation as its expected value is 0. However, past forecast errors can be quantified a posteriori by $\varepsilon_{t-1} = x_{t-1} - \hat{x}_{t-1}$.

Holt-Winters smoothing:

$$\hat{x}_{t+1} = l_t + b_t$$

The Holt-Winters forecast model employs double exponential smoothing (DES) that takes into account trends in the time series not being considered

3.4. Performance Anomaly Detection

by SES. The forecast value \hat{x}_{t+1} is made up of a smoothed level l_t and a current trend b_t for time t . Level and trend are calculated as follows:

$$l_t = \beta x_t + (1 - \beta)\hat{x}_t, \quad l_0 = x_0$$

$$b_t = \gamma(l_t - l_{t-1}) + (1 - \gamma)b_{t-1}, \quad b_0 = \Delta x_1$$

The model parameters are the data smoothing factor β ($0 < \beta < 1$) and the trend smoothing factor γ ($0 < \gamma < 1$). Moreover, the above Holt-Winters model can be extended by an additive or multiplicative third model parameter to allow for seasonal variations in the data.

ARIMA(1,0,1):

$$\hat{x}_{t+1} = c + \varphi x_t + \theta \varepsilon_t, \quad \varepsilon_t = x_t - \hat{x}_t$$

The ARIMA(1,0,1) model combines a first-order autoregressive process AR(1), i.e. $X_t = c + \varphi X_{t-1} + \varepsilon_t$, and a first-order moving average process MA(1), i.e. $X_t = c + \theta \varepsilon_{t-1} + \varepsilon_t$. The forecast model contains three model parameters: a constant term c , an autoregressive factor φ , and a moving average factor θ .

ARIMA(1,1,1):

$$\hat{x}_{t+1} = x_t + \varphi \Delta x_t + \theta \varepsilon_t, \quad \varepsilon_t = x_t - \hat{x}_t$$

The ARIMA(1,1,1) is an integrated variant of the ARIMA(1,0,1) model. Instead of the past values x_t , the first-order differences Δx_t are used in the autoregressive part. The underlying process model equation is $\Delta X_t = \varphi \Delta X_{t-1} + \theta \varepsilon_{t-1} + \varepsilon_t$. The model parameters are φ and θ .

2. Anomalous behavior hypothesis test: For anomaly testing, it is leveraged that not every individual operation execution has to be classified and reported if it is considered to be an outlier. Instead, it is rather desired that a cohesive collection of suspicious operation executions is observed before reporting an anomaly. In this way, the approached anomaly scoring aims at discovering a combined occurrence of contextual and collective anomalies. The proceeding benefits from the central limit theorem:

3. Self-Adaptive Performance Monitoring Approach

Whenever a random experiment is replicated many times, the new variable which equals the average result over the replicates is likely to follow the normal distribution.

Central Limit Theorem cf. [Montgomery and Runger, 2006]

Let X_1, X_2, \dots, X_n be a random sample of size n from a population with mean μ and finite variance σ^2 , i.e. the X_i 's are independent and identically distributed random variables.

Then for large n , the sample mean \bar{X} is approximately normally distributed with mean μ and variance σ^2/n . The limiting form of the distribution

$$Z = \frac{\bar{X} - \mu}{\sigma/\sqrt{n}}$$

is the standard normal distribution, as $n \rightarrow \infty$.

The sample mean \bar{x} of multiple response time observations $\vec{x} = (x_1, x_2, \dots, x_n)$ related to an operation in a specific stack context will be approximately normally distributed, irrespective of the shape of the common probability distribution from which the individual x_i 's have arisen from. So, no assumptions have to be made towards the distribution of the underlying population. The only premise is that the averaged observations form a random sample. Admittedly, it can be questioned if this is really the case, as the observations come from a time series.

To illustrate the central limit theorem, Figure 3.11 shows two series of histograms depicting frequency distributions of average response times. Each series demonstrates that the average response time of a specific operation (A or B , respectively) approaches to be normally distributed when the conjoint sample size increases (from 1 to 10 in both cases).

The distribution for individual executions of operation A shows two peaks, see (a1) in Figure 3.11. It exemplifies an operation having two major interior branches with different computing effort and thus bipartite timing behavior.

3.4. Performance Anomaly Detection

Which one of the branches is taken might depend on a boolean input parameter, for example.

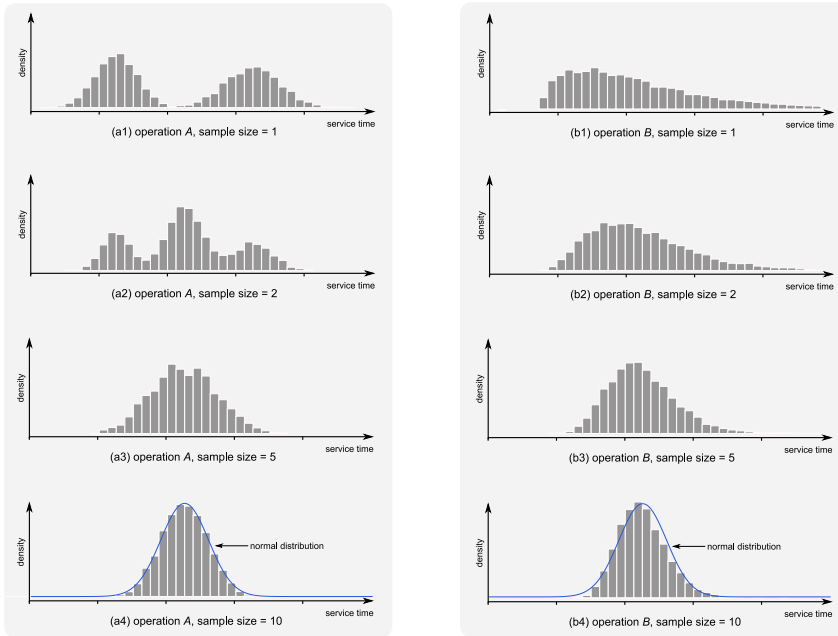


Figure 3.11. Frequency distributions of avg. response times for different sample sizes

The distribution for individual executions of operation B is clearly right-skewed, see (b1) in Figure 3.11. For instance, the long tail to the right might be an effect of varying system workload that influences the operation's timing behavior but has not been filtered out as a context-determinant dimension.

With a sample size of $n = 10$, the average response time of both distributions is fairly normalized. Therefore, this anomaly scoring algorithm bundles all operation executions with the same context that are observed in a specified

3. Self-Adaptive Performance Monitoring Approach

time interval (e.g. in 1 s) and draws a random sample of those with $n \geq 10$ if possible.

To decide if an anomaly is observed or not, a hypothesis test (Student's t-test) is conducted. It is tested whether the forecasted response time \hat{x}_{t+1} can be accepted as the mean of the population μ from which the sample has been taken. The validity of the test rests on the assumption that the sample mean \bar{x} is approximately normally distributed. The null hypothesis is $H_0 : \mu = \mu_0$ with μ_0 being the forecasted response time \hat{x}_{t+1} . The two-sided alternative hypothesis is $H_1 : \mu \neq \mu_0$. So, only if H_0 is rejected, the observed collection of response times is rated as anomalous. H_0 will be rejected if μ_0 lies in a critical region outside a confidence interval, as illustrated in Figure 3.12.

The range of the confidence interval depends on a specified significance level α . A fixed significance level, e.g. $\alpha = 5\%$, allows controlling the rate of false positives (probability of type I errors). The confidence coefficient $1 - \alpha$ expresses the probability that the confidence interval will contain μ . That is, the $(1 - \alpha)$ -confidence interval describes an interval with lower bound b_l and upper bound b_u satisfying $P(b_l \leq \mu \leq b_u) = 1 - \alpha$. If H_0 is true, the test statistic

$$Z_0 = \frac{\bar{X} - \mu_0}{\sigma / \sqrt{n}}$$

follows the standard normal distribution. As the population's standard deviation σ is unknown, it is estimated by the sample standard deviation S . Thus, the test statistic

$$T_0 = \frac{\bar{X} - \mu_0}{S / \sqrt{n}} \quad \text{with} \quad S = \sqrt{\frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n - 1}}$$

has a t-distribution with $n - 1$ degrees of freedom. By consequence, H_0 will be rejected if the value of the test statistic t_0 falls in the critical region defined by the lower and upper $\alpha/2$ percentage points of the corresponding t-distribution. The lower bound of the confidence interval is $b_l = -t_{1-\alpha/2, n-1}$, the upper bound is $b_u = t_{1-\alpha/2, n-1}$ (see Figure 3.12).

3.4. Performance Anomaly Detection

That is, reject H_0 if $|t_0| > t_{1-\alpha/2, n-1}$, which is the $(1 - \alpha/2)$ -quantile of t-distribution with $n - 1$ degrees of freedom [Montgomery and Runger, 2006].

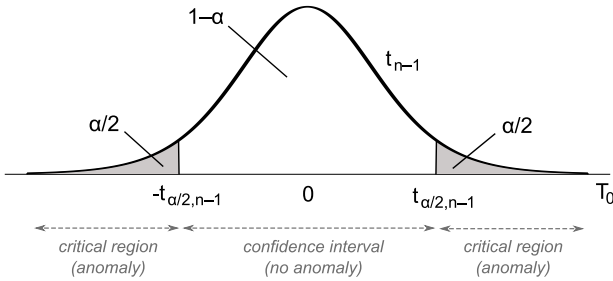


Figure 3.12. Confidence interval for anomaly hypothesis test (Student's t-test)

If t_0 is in the confidence interval, the recent sample of response time observations \vec{x} is associated with an anomaly value $a_{\vec{x}} = 0$. Otherwise, the sample is rated as anomalous, i.e. $a_{\vec{x}} = 1$.

3. Anomaly score calculation:: It is not sufficient to save the observation time of each collective anomaly. Instead, it is desired to calculate a single numeric measure that represents the recent degree of an operation's timing behavior to be anomalous. Therefore, an anomaly scoring function a is constructed that condenses the frequency and the trend of anomaly observations over time. For each operation, the function maintains an anomaly score between 0 and 1 where 0 indicates that response times have throughout been as expected, and 1 indicates a completely anomalous timing behavior. Each sample of response times \vec{x} that is tested for anomaly slightly impacts the overall anomaly score of an operation a_{op} , which is calculated recursively by exponential smoothing as follows:

$$a_{op,t} = \delta a_{\vec{x}} + (1 - \delta) a_{op,t-1}$$

The smoothing parameter δ determines how sensible the scoring function reacts. If anomalies are detected frequently, the score increases. Otherwise,

3. Self-Adaptive Performance Monitoring Approach

it will slowly decrease. An evaluation of the scoring function is presented in Section 5.2.

4. Anomaly score aggregation and correlation: The operation-level anomaly scores are aggregated to higher levels of abstraction such as component-level anomaly scores. Aggregation is done via weighted averages based on operation call frequencies. Further, the option is provided to adjust the anomaly scores by applying a correlation algorithm. On each level of abstraction, the system entities such as operations, classes, or components are represented as nodes of a call graph. The graph's directed edges represent the calling actions or dependencies among the system entities. It is commonly assumed that anomalies are propagated backwards through the call graph [Steinder and Sethi, 2004; Gruschke, 1998], i.e. if a node indicates anomalous behavior, then the anomaly is partially propagated to the node's callers. Correlation algorithms try to perform a negation of the propagation effects to identify the root cause of an anomaly. Hence, a node's anomaly score is adjusted by evaluating the propagation effects from its direct neighborhood or in its forward and backward transitive closure. For instance, a node's anomaly score is decreased if one of its callees has an even greater score, which probably indicates an anomaly backpropagation from the callee to the dependent caller node. Appropriate correlation algorithms are explicated in [Marwede et al., 2009].

Distribution-Based Anomaly Scoring

In this section, an alternative anomaly scoring procedure is presented that is based on clustering an observed distribution of response time samples. It consists of the following four steps, which are explicated in detail below.

1. Update an observed distribution of recent response time samples.
2. Apply an outlier detection algorithm that rates the probability of each distribution value to be a local outlier.
3. Calculate an anomaly score for each operation based on the outlier probabilities from 2.

3.4. Performance Anomaly Detection

4. Aggregate and correlate anomaly scores from 3.

1. Response time distribution update: For each operation, the distribution of observed response times is recorded. In the long run, it is not possible to store all response time samples in memory. Thus, the capacity C of the buffered distribution values is limited, with the buffer being a time-indexed rolling FIFO queue, i.e. if a new value is inserted and the capacity limit is reached, the oldest value is removed implicitly. Accordingly, only observations from the recent past are available to detect anomaly patterns. To stretch the history of the available data, multiple response time samples within a specified time interval (in the scale of a few seconds) are batched and aggregated. This proceeding supports the aim to detect collective anomalies, as described in 3.4.2. For each time batch, the median value \tilde{x} of the batch is inserted into the distribution values. It is not required to take the mean value, as it is done in the previous anomaly scoring algorithm from Section 3.4.3 for statistical correctness of an underlying t-test. The median is a more robust statistic than the mean, when it is desired to ignore heavy point anomalies while searching for collective anomalies.

2. Distribution-based computation of local outlier probabilities: The observed distribution of median values $\vec{\tilde{x}}$ serves as input for an outlier detection algorithm called *local outlier probabilities (LoOP)* by Kriegel et al. [2009]. LoOP is a recent extension of the *local outlier factor (LOF)* algorithm [Breunig et al., 2000]. Given a set of input records, LOF is based on the concept to compare a record's local density with the local density of its neighbors. Local density is understood as a distance, at which a record can be reached from other records in its locality, i.e. from its k nearest neighbors. Typically, LOF and LoOP are used to detect outliers in sets of multidimensional records, e.g. space coordinates. Figure 3.13 illustrates the concept of LoOP.

In case of a distribution of median response times $\vec{\tilde{x}}$, the record set is one-dimensional. The local density of each value \tilde{x} is compared to the densities of its neighbors (considering locality in the distribution values independent of observation time). Neighboring values should have similar

3. Self-Adaptive Performance Monitoring Approach

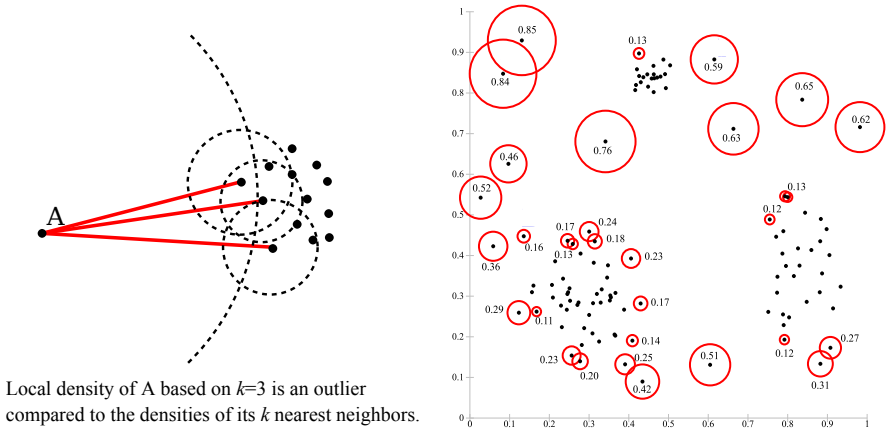


Figure 3.13. Local outlier probabilities (LoOP) [Kriegel et al., 2009]

local densities. Otherwise if a value has a substantially lower density than its neighbors, it is considered to be an outlier. Each value is assigned with an outlier factor quantifying the distance to its neighborhood. With LoOP, this resulting outlier factor is transformed to a range of $[0; 1]$ being interpretable as the probability of a time batch median value $a_{\bar{x}}$ to be an anomaly.

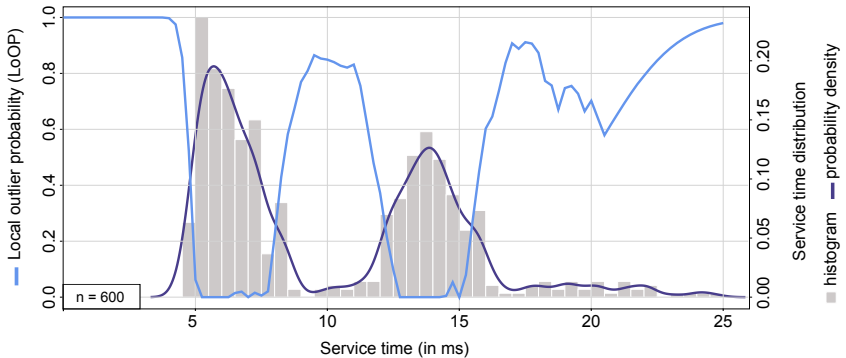


Figure 3.14. LoOP applied to operation response time distribution

3.4. Performance Anomaly Detection

Figure 3.14 depicts the response time distribution of a synthetic operation in form of a histogram and a derived probability density curve. The operation's timing behavior has been observed for 30 min. with a time batch interval of 3 s, so that 600 median values are taken into account. The response time distribution shows two regions which most of the median values fall into. The overlaid LoOP curve indicates that the LoOP algorithm assigns very low anomaly probabilities to these high-density regions, while values outside and between the regions are likely to indicate anomalies.

3. Anomaly score calculation: The continuous operation-level anomaly score a_{op} is derived as a weighted moving average over the recent distribution-based anomaly probabilities $\{a_{\bar{x}}\}$. The moving average (of size w) is ordered decreasingly by observation time index i , so that newer anomaly values are weighted more heavily than older values:

$$a_{op} = \frac{\sum_{i=1}^w i a_{\bar{x}_i}}{w(w+1)/2}, \quad 1 \leq w \leq C$$

The moving average window size w and the time batch interval for median retrieval determine how far the anomaly score a_{op} looks back into the past.

4. Anomaly score aggregation and correlation: This optional step is identical to the last step of the time series-based anomaly scoring procedure, as described in Section 3.4.3.

The implementation of the two presented anomaly scoring approaches as part of the Kieker Analysis tool is described in Section 4.3.6. Different parametrization variants of both approaches are evaluated in Section 5.2.

Continuous Software System Monitoring Integration

In this chapter, a proof of concept for continuous and adaptive software system monitoring is provided by means of the design and implementation of the Kieker monitoring framework. The software-technical implementation is essential to demonstrate the feasibility of the approach and builds the basis for the later experimental evaluation. Section 4.1 presents the Kieker Monitoring component, whereas the Kieker Analysis component is described in the following Sections 4.2 (analysis plugin subscription model), 4.3 (analysis visualizations) and 4.4 (MonitoringAdaptation plugin).

As the Kieker project initially targeted monitoring of Java EE-based systems, it has consequently been implemented in Java itself. Its design is established by means of EMF meta-models that allow generating major parts of the Java code. Thus, the Kieker project follows the principle of model-driven software engineering.

4.1 Monitoring Agent Design

Instances of the Kieker Monitoring component are called *monitoring agents*. The internal realization of an agent is subdivided into six specific packages: control, system, instrumentation, datacollection, record, and persistence. The

4. Continuous Software System Monitoring Integration

scope of each package is shortly explicated in the following. The dependencies among the packages and a selection of their immanent classes are depicted in Figure 4.1. The figure covers only those classes that are referenced in the following.

The agent's controller (package control): As described in Section 3.2.2, a monitoring agent is deployed at each execution container of the monitored software system. That is, there will be as many active monitoring agents as application server instances to which the system components are distributed across. All monitoring agents can be controlled remotely by a single instance of the Kieker Analysis component.

The control package contains the classes that realize the Monitoring Controller component and its provided Adaptation Interface depicted in Figure 3.2. A class diagram of the package is shown as part of Figure 4.1. The package includes the MonitoringController singleton class and its associated MonitoringRemoteService class, which implements a broad remote interface (being denoted generically as Adaptation Interface in Figure 3.2) and thus allows external control over a monitoring agent. So far, the MonitoringRemoteService is derived from Java's Remote Method Invocation (RMI) interface `java.rmi.Remote`, which serves to identify interfaces whose methods can be invoked from any non-local virtual machine [Sun Microsystems, Inc., 2004]. The interface methods provided by the MonitoringRemoteService class allow:

- querying configuration properties of the monitoring agent, e.g. how to connect to its Monitoring Log into which the collected records are pushed,
- querying information about the static structure of the monitored system, particularly its components, classes, and operations,
- querying information about the runtime container of the monitoring agent, e.g. attributes of the underlying JRE, OS, and CPU,
- starting/stopping a background thread that enables continuous observation and collection of resource utilization data, e.g. the CPU load,

4.1. Monitoring Agent Design

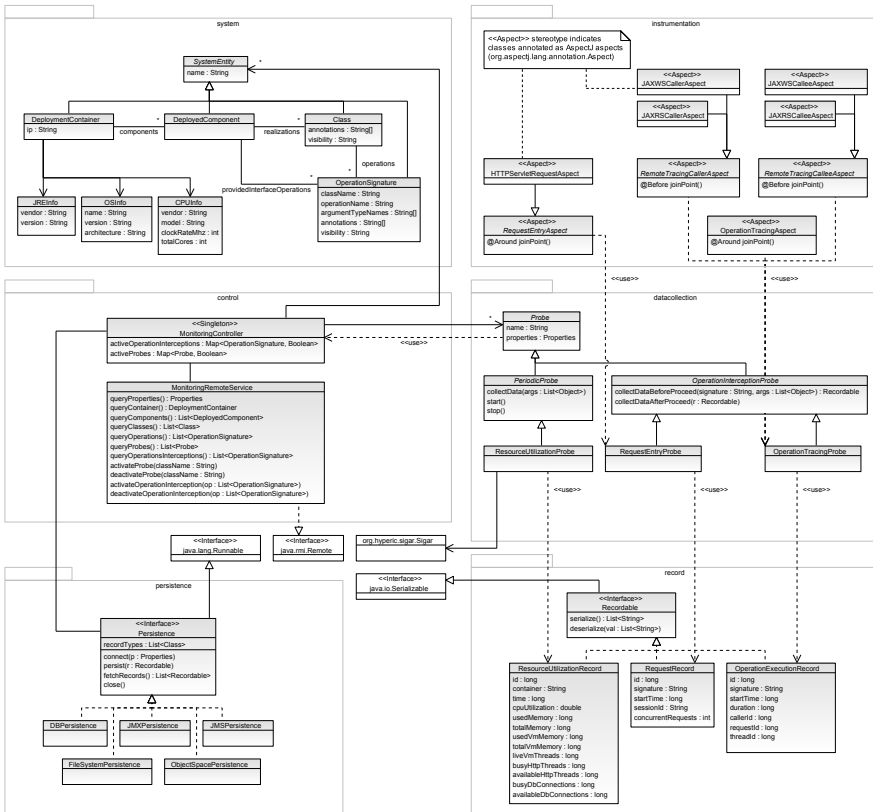


Figure 4.1. Class diagram of the Kieker Monitoring component realization

- activating/deactivating probes or measuring points according to passed signatures.

The agent's system model (package system): The system package contains classes to describe the static structure of the monitored software system, made up of different types of system entities. Specific system entities are components, classes, and operations. For each system entity type, an

4. Continuous Software System Monitoring Integration

agent's controller (class `MonitoringController`) manages a list of all known system entity objects, e.g. a list of all known classes. An agent's controller is instantiated when one of the instrumented probes is triggered for the first time. In case that the monitored system is distributed over multiple execution containers, each monitoring controller will only get to know those system entities being deployed at the container it monitors. An integrated view on the entire software system will be put together in the later analyses that merge observation data from all monitoring agents. As no static code reverse engineering is conducted, only those system entities become known that are executed during runtime. For components and classes, this means that they become known after any enclosed operation has been executed.

Classes and operations can easily be extracted using the Java Reflection API. Unfortunately, this is not possible for components. At runtime, it cannot clearly be extracted which classes (or subcomponents) realize a component and which operations are part of a component's provided interface. Java does not support component declaration by syntactical keywords. The current Java EE specification [Sun Microsystems, Inc., 2009] provides a very cloudy definition of what is a component in Java. The specification names four types of components: (1) application clients, typically GUI programs that execute on client devices, (2) applets, GUI components that execute in a client-side browser, (3) web components based on Servlets¹ (or higher-level abstractions such as JavaServer Faces² or JavaServer Pages³) that typically respond to HTTP requests and generate HTML or XML output, and (4) EJBs that execute in a server-side transactional context and usually contain the application logic. These component types allow diverse ways to provide interfaces for their clients, e.g. via managed bean operations that are directly addressed from a GUI, via the EJB remote interface (`javax.ejb.Remote`)⁴, via web services that may either be SOAP-based (JAX-WS)⁵ or REST-based (JAX-RS)⁶, via RMI, etc. These various contemporary technologies will

¹ Java Servlet 3.0 Specification: <http://www.jcp.org/en/jsr/summary?id=315>

² JavaServer Faces 2.0: <http://www.jcp.org/en/jsr/summary?id=314>

³ JavaServer Pages 2.1: <http://www.jcp.org/en/jsr/summary?id=245>

⁴ Enterprise JavaBeans (EJB) 3.0: <http://www.jcp.org/en/jsr/summary?id=220>

⁵ Java API for XML-Based Web Services (JAX-WS): <http://www.jcp.org/en/jsr/summary?id=224>

⁶ Java API for RESTful Web Services (JAX-RS): <http://www.jcp.org/en/jsr/summary?id=311>

4.1. Monitoring Agent Design

keep evolving. Therefore, it seems currently not be efficient and lasting to develop a method that tries to automatically detect component realizations and interfaces at deployment time or runtime. In the following, such a method is regarded as future work.

So far, it is assumed that a more goal-oriented solution is to adopt intensively studied component identification strategies based on object-oriented cohesion and coupling metrics [Lee et al., 2001; Birkmeier and Overhage, 2009; Arisholm et al., 2004; Briand et al., 1999; Koschke, 1999]. The referenced component identification strategies help to cluster the classes and to assign them to components. Afterwards, the component interfaces incorporate those operations that are called from outside its assigned component. In order to allow component-based analyses, the findings have to be made available to the Kieker Analysis instance. An appropriate way to achieve this is to inject component-declaring annotations by means of AOP into the monitored software system. To give an illustrating example, the annotations `@ComponentRealizationPart` and `@ComponentInterfaceMethod` are introduced in Listing 4.1. At the top of the listing, a class is exemplified that is annotated to be part of a component `c1` and that contains a method `m1` being part of `c1`'s provided interface. At the bottom, an aspect class is shown that allows separating the annotations from the original code and to weave them into the code subsequently.

If model-driven code generation is already employed to create parts of the monitored system's code, it is alternatively feasible to extend the system model and the code generator to support the injection of the component-declaring annotations.

A further system entity in the system package is the monitored execution container itself at which the components are deployed. As shown in Figure 4.1, a container provides information about its IP and the underlying JRE, OS, and CPU.

4. Continuous Software System Monitoring Integration

```
@ComponentRealizationPart(component="c1")
class MyClass {

    @ComponentInterfaceMethod(component="c1")
    public void m1() { /* do... */ }

    void m2() { /* do... */ }
}

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareAnnotation;

@Aspect
class DeclareAnnotationAspect {

    @DeclareAnnotation("MyClass")
    @ComponentRealizationPart(component="c1") Object cl;

    @DeclareAnnotation("public * MyClass.*(..)")
    @ComponentInterfaceMethod(component="c1") void m() {}
}
```

Listing 4.1. Component-declaring annotations

Data structures constructed and delivered by the agent (package record):

The record package contains derivative classes of the Recordable interface, e.g. RequestRecord, ResourceUtilizationRecord, and OperationExecutionRecord. The Recordable interface extends java.io.Serializable and prescribes methods to serialize and deserialize record objects. The instrumented and activated probes instantiate, fill, and persist record objects into a record repository called Monitoring Log (see Figure 3.2). The records will be consumed from the repository by log readers of the Kieker Analysis instance. In the record package depicted in Figure 4.1, three exemplary record types are listed, which are described in the following enumeration. As shown, they are used by corresponding probes from the datacollection package. The record package is easily extendable by new record types that implement the Recordable interface.

- RequestRecord: Represents an incoming client request at the system interface, e.g. via a HTTP Servlet invoked through a web-based GUI. A request can be referenced by multiple OperationExecutionRecords being created while the request is processed. The record saves the number of related OperationExecutionRecords, the start time of the request, the

4.1. Monitoring Agent Design

number of requests being concurrently executed, and available session information such as the related HTTP session.

- **OperationExecutionRecord:** Represents a single operation execution within a request's call trace. The record saves the operation's signature (derived from the join point via reflection), the execution's start time and duration (in nanoseconds), the executing thread, a reference to the request (id of RequestRecord) of which the operation execution is part of, and its parent caller execution in the request trace. The information about the caller is buffered in thread local memory for non-remote calls, or attached to and extracted from the meta-data in case of remote calls. The information embedded in the OperationExecutionRecord objects allows for the analysis of call traces and operation timing behavior.
- **ResourceUtilizationRecord:** Represents a discrete observation of the surrounding resource utilization. The record saves metrics such as the current CPU utilization in %, the amount of used and available memory at OS- and VM-level in MB, the number of busy and free threads and database connections as managed by the application server. Besides, the record saves a time of observation and a reference to the execution container to which the metrics are related to. Further utilization metrics can be added (see description of ResourceUtilizationProbe below).

Data logging (package persistence): The persistence package provides alternative writers to log the records into different storage repositories for data exchange between the monitoring agents and the Kieker Analysis instance. As the writing operations provoke a major part of the measurement cost (see Section 5.3.1), the writers work asynchronously. That is, a concurrent writer thread is launched in order not to delay the monitored application significantly. Each writer implements the Persistence interface. As shown in the persistence package in Figure 4.1, the interface prescribes four methods: (1) connect to a Monitoring Log, (2) close an established connection, (3) persist a Recordable object into the log, and (4) fetchRecords currently stored in the log, returning a list of Recordables. The connect

4. Continuous Software System Monitoring Integration

and close methods are used by both, the monitoring agents as well as the analysis instance.

The method to persist records is called from the probes of the monitoring agents. As stated above, the different implementations of the persist method do not write the records directly into the underlying Monitoring Log. Instead, they are pushed into a temporary in-memory buffer, from where they are read and written to the log by a concurrent writer thread. The asynchronous record writing behavior of the Persistence implementations is illustrated in the sequence diagram depicted in Figure 4.2, where a concurrent writer thread is notified from the persist method.

The method fetchRecords is used by the log readers of the analysis instance. A Persistence implementation manages a list of all known record types that have ever been persisted. The record type list can be queried via the Monitoring Controller's remote interface. This enables the log readers to know in advance which record types rest inside a log.

In the following, the implemented persistence providers are listed. Which persistence provider is instantiated and used as a Monitoring Log is decided when the Monitoring Controller is initiated and reads in its configuration properties.

- `FileSystemPersistence`: The records are written to flat files on the local or a remote file system. A configuration property specifies in which directory on the file system the records are to be stored. The default file format are comma-separated values, with each record being saved in a single line of a file. Each time a configured interval has elapsed (e.g. 3 s), a new file is generated that contains all records having been collected in that interval. Files that have been consumed by a log reader can implicitly be deleted, particularly in case of continuous online analysis. Apart from that, the `FileSystemPersistence` is as well appropriate for later offline analysis, with longer time intervals per file and disabled file deletion.
- `DBPersistence`: The records are written to a relational database. A configuration property specifies how to establish the JDBC connection

4.1. Monitoring Agent Design

(driver class, protocol, host, port, database schema, user credentials). The database tables and their constraints are created after the connection has been opened. For each record type, a separate table is created. Each record is saved as a tuple with its id as a unique key value. When a record is consumed by a log reader, it is deleted from its table or marked as read. For efficient insertion of record bulks, precompiled SQL statements (class `java.sql.PreparedStatement`) are applied. Such prepared statements provide the benefit of not having to be parsed and validated each time they are executed. By consequence, multiple SQL inserts into the record tables that only differ in a few parameter values can efficiently be executed in a batch. As there exist minor differences between the database system vendors' SQL dialects (e.g. concerning the DB type system), vendor-specific implementation variants such as `OracleDBPersistence` or `MySQLDBPersistence` are provided.

- `JMSPersistence`: The records are written into message queues as specified by the Java Message Service (JMS) API. A configuration property specifies the address of the used message service broker and queue. Each record is transferred as a serialized object using the `javax.jms.ObjectMessage` interface. The required message service broker can either be run as a stand-alone service or can be embedded into the application server that already deploys the monitored system. An advantage of the `JMSPersistence` is that the JMS protocol is conceived for asynchronous remote communication and supports the observer pattern. That is, a log reader does not have to poll periodically for new records, but will be informed as a subscriber when new records are published by the monitoring agents' writer threads.
- `JMXPersistence`: The records are published via a brokering agent called `MBeanServer` according to the JMX API. The `MBeanServer` is typically run by the Java EE application server that hosts the monitored system components. Essentially, the `JMXPersistence` provider is a `MBean`, i.e. a JMX probe being registered at the `MBeanServer`. For each record to be transferred, a notification is emitted by this `MBean` containing the serialized record in its data part (see

4. Continuous Software System Monitoring Integration

`javax.management.NotificationEmitter` interface). The Kieker Analysis component and any other JMX client, which is authorized to connect to the MBeanServer, can subscribe itself to receive the published records. Like JMS queues, JMX notifiers realize the observer pattern. A difference with JMX is that records are lost if no subscriber is registered.

- `ObjectSpacePersistence`: The records are written to an object-oriented tuple space based on the Linda coordination language [Gelernter, 1985]. A configuration property specifies how to connect the used object space service. As today no Java API for tuple spaces can be considered as a common standard, the space-based persistence implementations are vendor-specific. The provided implementation uses the Fly Object Space⁷ library. As object spaces are conceived for efficient coordination of concurrent processing in distributed systems, they keep their input data in memory and do not write it to persistent storage. By consequence, a space-based persistence provider is only suitable for online analysis, not for offline analysis. When a record is pushed into the object space, it is marked with a time to live in the space. If it is not consumed until its expiry, it will be removed without having been transmitted to any analysis. The provided implementation also allows publish and subscribe according to the observer pattern.

The database-, JMS-, and object space persistence providers guarantee transactional integrity according to their inherent default protocol. To extend the above list, new implementations of the Persistence interface can be added easily. Asynchronous record writing is reasonable, but not required.

Instrumentation of probes (package instrumentation): The instrumentation package enfolds generic and derived technology-specific aspects to inject probes into the monitored system. The Kieker Monitoring component provides default means for AOP-based instrumentation as described in Section 2.3. Though AOP is the recommended instrumentation technique, this does not

⁷ <http://www.flyobjectspace.com/>

4.1. Monitoring Agent Design

exclude alternative or complementary techniques, as for example manual probe injection. The following generic aspects are provided:

- `RequestEntryAspect`: Identifies the measuring points in the monitored system where client requests enter the system. A concrete implementation of this aspect is the `HttpServletRequestAspect`, which handles incoming HTTP requests. It aims at intercepting the Servlet API method `javax.servlet.Servlet.service(..)`, and thus is woven into the particular derived Servlet class of the used web application framework, e.g. `javax.faces.webapp.FacesServlet` in case of JSF, `org.apache.struts.action.ActionServlet` in case of Apache Struts, or `net.sourceforge.stripes.controller.DispatcherServlet` for Stripes.
- `OperationTracingAspect`: Identifies the measuring points of all operations of the monitored system that are in scope of monitoring. The monitoring scope has to be defined via pointcut expressions in the AOP configuration file. Typically, weaving the `OperationTracingAspect` should be restrained to the original implementation parts of the monitored system itself. That is, the interior behavior of any addressed third-party libraries, frameworks, external services, and the runtime environment would not be in the scope of monitoring. Appropriate ways to achieve this are:
 - Application-specific pointcuts: An application-specific pointcut expression restrains aspect weaving to only those methods that match a particular pattern, e.g. `execution(* org.spec.jent.*(..))` to intercept the methods in the package trunk of the SPECjEnterprise2010 benchmark.
 - Monitoring-specific annotations: A monitoring-specific annotation such as `@Monitored` is manually added to each method that should be instrumented. Afterwards, the pointcut expression references all methods that are decorated with this annotation via `@Monitored **(..)`.

The pointcuts of the `OperationTracingAspect` and the `RequestEntryAspect` are associated with an around-operation advice, i.e. additional aspect code is woven before and after the original bodies of the intercepted operations being addressed by the pointcut. At load-time of a class

4. Continuous Software System Monitoring Integration

being affected by a pointcut, the aspect weaving procedure injects the code of the around advice into the original class' byte code. This is illustrated in Figure 4.2: An intercepted operation of an instrumented class is called. The execution of the original operation body is invoked by the call of `proceed()` in the lifeline of the instrumented class. Before and after, calls to a monitoring probe are made. As shown in Figure 4.2, the probe is an instance of the `OperationInterceptionProbe` interface being explicated below.

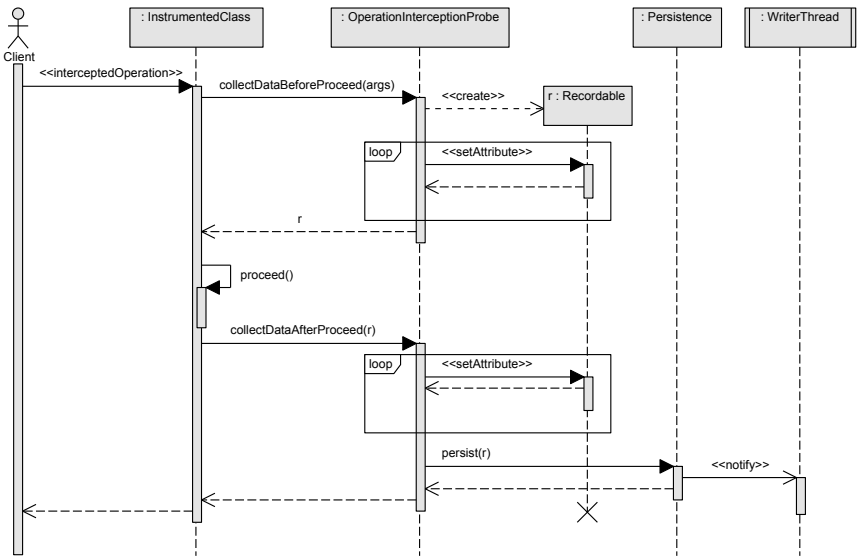


Figure 4.2. Sequence diagram related to the operation interception probe

- `RemoteTracingCalleeAspect` and `RemoteTracingCallerAspect`: Identifies the measuring points of remote calls in the monitored system. As there exist several protocols (again with several implementations) that can be applied for remote communication, specific caller and callee aspects for remote tracing have to be written. As shown in the instrumentation package in Figure 4.1, the Kieker Monitoring component provides remote tracing aspects for web services built with JAX-WS

4.1. Monitoring Agent Design

or JAX-RS, for example. If the monitored system makes use of these common technologies, the related aspects provide an appropriate means for instrumentation. Otherwise, the generic aspects serve as an extension point for other technologies that are not covered by the delivered aspect set.

Each caller aspect intercepts the client-side endpoint of a remote call, while a callee aspect intercepts the related server-side endpoint. In case of JAX-WS (version 2.0), these are for example: client-side calls to `javax.xml.ws.Service.getPort()`, and server-side calls of methods annotated as `@javax.jws.WebMethod`. To enable remote tracing, a caller aspect adds information, namely the current `RequestRecord` id and the caller's `OperationExecutionRecord` id, to the meta-data of the remote call, e.g. into the SOAP- or HTTP header in case of JAX-WS. This tracing information is processed by the callee aspect. Preliminary remote tracing approaches can be found in [Sahai et al., 2002; Parsons et al., 2008; Meyerhöfer, 2007].

It is appropriate to define the aspect pointcuts in a separate configuration file. This way, the Java sources have not to be compiled each time a monitoring agent is configured and packaged for a specific software system. An example configuration file for AspectJ (typically called "META-INF/aop.xml") that instruments all methods of the SPECjEnterprise2010 benchmark with the `OperationTracingAspect` is illustrated in Listing 4.2.

4. Continuous Software System Monitoring Integration

```
<?xml version="1.0" encoding="UTF-8"?>
<aspect>
  <aspects>
    <concrete-aspect name="concrete-aspect_1"
      extends="kieker.monitoring.instrumentation.OperationTracingAspect">

      <pointcut name="pointcut_1" expression="execution(* org.spec.jent..*(..))"/>
    </concrete-aspect>
  </aspects>

  <weaver options="-showWeaveInfo">
    <include within="org.spec.jent..*" />
  </weaver>
</aspect>
```

Listing 4.2. AspectJ pointcut configuration example

The packaged archive named `kieker.monitoring.jar` that manifests a monitoring agent has to be added to the application servers' classpath. When an application server starts, this archive will be processed by the AspectJ weaver (`aspectjweaver.jar`), which has to be set up as a Java agent (see Section 2.3) in advance.

Okanovic et al. [2011] describe the possibility to integrate the aspect configuration file into a deployed component artifact and modify this file during runtime. This technique causes a change of the artifact's timestamp, and thus the application server to redeploy the component and re-weave the aspects. Possible session losses and transactions breaks are serious drawbacks why this adaptive instrumentation technique is not pursued.

Data collection by probes (package `datacollection`): The `datacollection` package contains the monitoring probe types. The probe instances are invoked from the above aspects or other instrumentation means. For each concrete probe type, a single instance is created and managed by the monitoring agent's controller (see package `datacollection` in Figure 4.1). The following list of provided probe types is extensible:

- `RequestEntryProbe`: Called from the `RequestEntryAspect` advice, creates and persists `RequestRecords`.

4.1. Monitoring Agent Design

- `OperationTracingProbe`: Called from the `OperationTracingAspect` advice, creates `OperationExecutionRecords` that are persisted with the `RequestRecord` which they are part of. Each `OperationExecutionRecord` has a reference (caller id) to an object of the same type, which represents its parent in the call stack (only the root call has none). As long as a request is executed by a single thread, the caller id can be saved to and extracted from thread-local memory (`java.lang.ThreadLocal<T>`). As stated above, to trace remote (or asynchronous) calls, the request and caller reference ids are attached to the call's meta-data. The concrete `RemoteTracingCalleeAspect` advices extract these reference ids and use it to initiate the server-side thread-local memory.

The `OperationTracingProbe` as well as the `RequestEntryProbe` implement the `OperationInterceptionProbe` interface which prescribes two methods for collecting monitoring data before and after the proceeding of the actual operation body: `collectDataBeforeProceed(List<Object>):Recordable` and `collectDataAfterProceed(Recordable)`. As indicated by the loop fragments in Figure 4.2, the before- and after-proceed methods particularly set the record attributes. Evidently, some values such as the operation start time have to be observed before-proceed, others such as the operation duration can only be observed after-proceed. The before-proceed method accepts a list of arbitrary input arguments to pass information about the current measuring point, e.g. its signature and parametrization derived from an AspectJ join point instance. It returns a new `Recordable` object being passed to the after-proceed method (see Figure 4.2). Finally, the after-proceed method persists the new record.

- `ResourceUtilizationProbe`: Called from the `ResourceUtilizationCollector` thread, creates and persists `ResourceUtilizationRecords`. Starting, stopping, and keeping alive the `ResourceUtilizationCollector` is delegated to the Kieker Analysis instance via the interface provided by the `MonitoringRemoteService`. If launched, the thread will periodically invoke the probe to collect OS- and VM-level resource utilization

4. Continuous Software System Monitoring Integration

statistics. The remote interface allows adapting the collection interval and a time-to-live parameter at runtime. The time-to-live parameter controls how long the `ResourceUtilizationCollector` thread continues though it is no longer kept alive by the Kieker Analysis instance.

Querying detailed information about the utilization of hardware resources is not possible from inside the JVM. Thus, native OS-specific tools have to be called via the Java Native Interface (JNI), which is bridging the gap between the virtual machine and its underlying platform. A native library for each platform (OS version, processor architecture) is required. The Kieker Monitoring component leverages the open source SIGAR API⁸, which is a cross-platform API for collecting OS-level resource data. It supports Linux, Windows, Mac OS X, Solaris, AIX, HP-UX and FreeBSD for a variety of versions and architectures. The observable resource data includes information about

- the system-wide or per-process state and load of memory, swap, and the CPUs,
- the file system and its load,
- the network interfaces and related statistics including network routes and connection tables.

To monitor VM-level software resources managed by a Java EE application server, the Kieker Monitoring component connects to a monitoring and brokering agent, the `MBeanServer` based on the JMX API, which is run by the application server. The `MBeanServer` supplies JVM runtime statistics and utilization data for thread pools, JDBC connection pools, etc. Addressing specific `MBean` probes is part of the Kieker Monitoring component's configuration properties, as it depends on the vendor of the application server and the monitored application itself.

By default, a `ResourceUtilizationRecord` provides fields to capture the utilization of a CPU, memory at OS- and VM-level, a thread pool, and a database connection pool. To forward further metrics if required, it can be extended or similar records types can be introduced.

⁸ <http://www.hyperic.com/products/sigar/>

4.2 Analysis Subscription Model

The Kieker Analysis component is conceived to be run as a singleton instance, which brings together and processes monitoring data from several distributed monitoring agents. A first step of setting up the monitoring process in the Kieker Analysis tool is to configure from which monitoring agents data will be consumed. This is done in the Data Provision subcomponent, whose data model is shown in the `dataprovision` package of the class diagram in Figure 4.3. In the design of the Kieker Analysis component, a monitoring agent is represented by the `MonitoringAgent` class. For connecting to a particular monitoring agent, its host, port, and RMI binding name have to be supplied. Given that information, a remote connection can be established via the agent's `Adaptation Interface` to initialize a client-side `MonitoringRemoteService` proxy. With this proxy, it is possible to execute remote operations on the monitoring agent as described in Section 4.1 (agent's control package). One of these remote operations allows querying information about the used `Monitoring Log`. Based on this information, a compatible `Persistence connector` is created that allows interaction with the `Monitoring Log`.

As shown in Figure 4.3, each monitoring agent is associated with a log reader (class `LogReader`). A log reader connects to an agent's `Monitoring Log` and fetches the records having previously been pushed into the log by the agent itself. Depending on the used log, the log reader is either notified about new records (e.g. JMS, JMX) or periodically polls for new records (e.g. file system, DB). A notification/polling interval to receive record batches is configurable. The model of the `Data Provision` component depends on the Kieker Monitoring model, as it references some of its elements such as the `Recordable` derivatives and the `Persistence interface`. A log reader forwards its fetched records to subscribed data processing filters. The subscription model between such analysis filters is described in the following.

The data stream throughout the Kieker Analysis component follows the pipes-and-filters pattern. Its underlying model is shown in the project

4. Continuous Software System Monitoring Integration

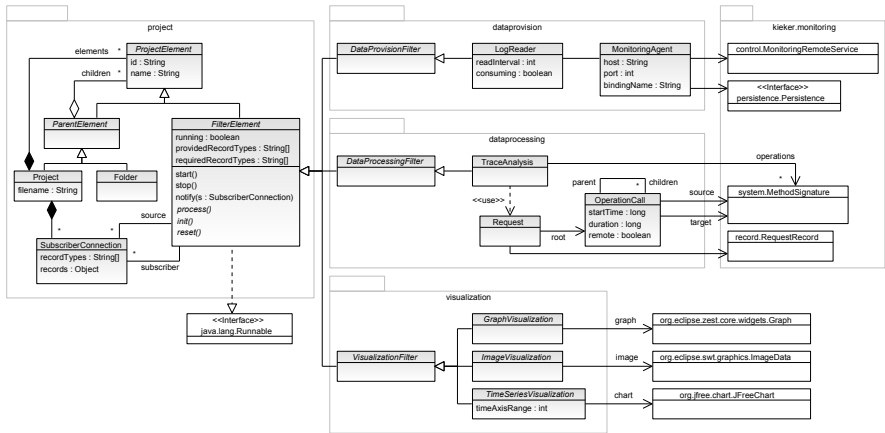


Figure 4.3. Class diagram of the Kieker Analysis component realization

package as part of Figure 4.3. All filters are derived from the FilterElement class, while the connecting pipes between the filters are represented by instances of the SubscriberConnection class. That is, filters can interactively be subscribed to other filters, given that at least one of the record types provided by the source filter matches the record types required by the subscriber filter. All filters and their connections are contained and managed in a project (class Project). Filters are specific project elements derived from the abstract ProjectElement class. Folders allow for further hierarchical structuring of the elements within a project. With the design model being based on EMF, a project serving as the root content of an EMF resource and its containments can easily be stored in an exchangeable XML or XMI representation via the EMF persistence framework.

Each filter runs its own thread that can be started and stopped via the control panel. The subscription model implements the observer pattern, i.e. an observable filter notifies all its subscribed filters when it pushes new records into its outbound pipes. A subscriber does not have to poll periodically for new records, but will be notified across the inbound pipes by the notify(SubscriberConnection) method. As indicated in Listing 4.3, a notified

4.2. Analysis Subscription Model

filter executes its abstract process method, which has to be implemented by each concrete FilterElement. A generic implementation pattern for the process method is:

1. Fetch the input records from the notifying inbound connection.
2. Process the input records to create or to update specific output records.
3. Deliver the output records to the outbound subscriber connections and notify the subscribers.

```
public abstract class FilterElement implements Runnable {  
  
    private boolean running;  
    private FilterElement notifier;  
  
    public abstract void process();  
  
    public void notify(SubscriberConnection s) {  
        notifier = s.getSource();  
        synchronized (this) { notify(); }  
    }  
  
    @Override  
    public void run() {  
        while (running) {  
            process();  
            synchronized (this) { wait(); }  
        }  
    }  
  
    // ...  
}
```

Listing 4.3. Kieker Analysis filter notification pattern

Log readers are placed at the beginning of the data stream throughout a Kieker Analysis project. The source of the data stream are the records continuously arriving from the logs of the monitoring agents. These records are forwarded to subscribed data processing filters. A basic data processing filter is a TraceAnalysis instance that takes RequestRecords from one or more log readers as its input. These records are processed to construct Request objects. Each Request object is an object-oriented representation of a single request's trace through the system. Figure 4.4 provides an illustrating example for a request call trace. A Request is associated with a tree structure of OperationCall objects by a pointer to the tree root, cf. Figures 4.3 and 4.4.

4. Continuous Software System Monitoring Integration

The tree structure of OperationCalls captures the complete monitored call trace of an incoming system request. Each OperationCall represents a single call of an arbitrary system operation. The called operation is referenced by the target attribute, while the source attribute references the operation from which the call is initiated. Besides, information about the start time and the duration of the operation execution is saved, as well as whether the target operation has been called remotely. In case of remote calls, OperationExecutionRecords from different monitoring agents are merged to construct a Request object.

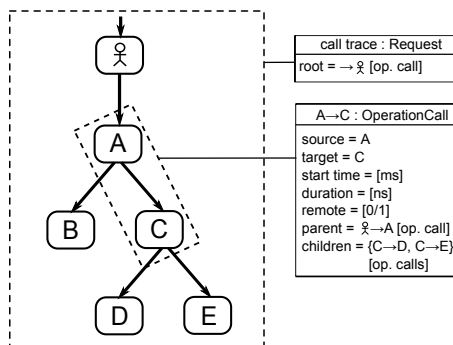


Figure 4.4. Illustration of a request call trace and an operation call

The Requests are forwarded to subsequent data processing filters that are subscribed to the TraceAnalysis. Hence, the TraceAnalysis is an auxiliary filter whose results are not presented to the user, but serve as input for higher-level filters. If there is no running subscriber filter, the outgoing requests are not saved and thus lost for further analysis. A selection of implemented higher-level analysis filters and their corresponding visualizations are described in the following section.

Figure 4.5 shows a screenshot of the Kieker Analysis tool with the project navigator in the left view part and a project stream editor for one of the example projects in the right view part. The project navigator contains all recently opened projects, having the project elements being structured hierarchically by folders. The project stream editor displays all filter

4.3. Visualization of Analysis Results

elements of a project and their interconnections in consequence of being subscribed to each other. New elements can be added via the project stream editor's palette, via the top menu, or via context-sensitive popup menus. Deletion and editing of project elements is available via the menus or via shortcuts, e.g. a double-click on any project element opens a specific editor to modify the selected element. The project stream editor indicates filter elements that are running by a green background color. In the state displayed in Figure 4.5, a TraceAnalysis filter named "Trace Analysis" is configured to consolidate records delivered by three LogReaders named "Log Reader (blade<1-3>)", of which only the first two are currently active. The TraceAnalysis filter sends its output exclusively to a subscriber named "Use Cases", as it is the only one of the three subscribed analysis filters being currently active. Each analysis filter has further subscribers, mainly specific visualizations.

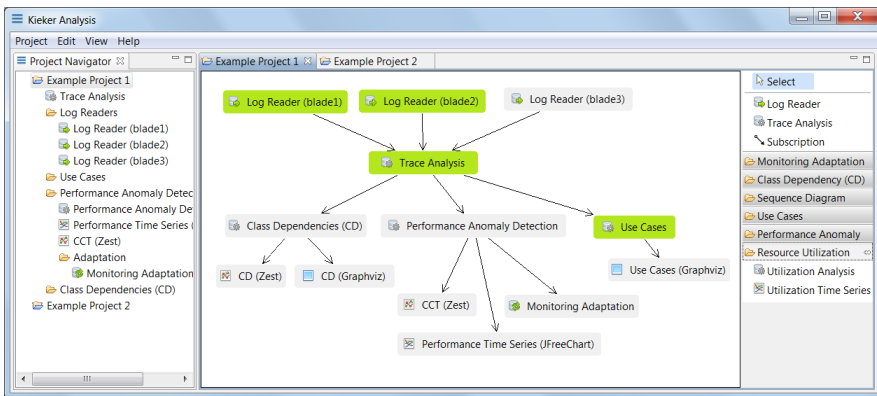


Figure 4.5. Screenshot of the Kieker Analysis project stream editor

4.3 Visualization of Analysis Results

The visualization package in Figure 4.3 indicates that the visualizations of the analysis results (derived from the Visualization class) are filter elements as

4. Continuous Software System Monitoring Integration

well. These visualizations are the sink of the data stream in a Kieker Analysis project. They do not provide output for further subscribers, but visualize the state of underlying analysis filters to the user of the Kieker Analysis tool. In dependence of their used libraries, different types of visualizations are:

- Interactive graphs (class `GraphVisualization`) that utilize Eclipse Zest, i.e. a graph visualization toolkit being based on Draw2d as part of the Eclipse Graphical Editing Framework (GEF)⁹.
- Static graph/diagram images (class `ImageVisualization`) that use third-party libraries such as Graphviz to generate images of graphs or specific diagrams, e.g. UML sequence diagrams.
- Time series graphs (class `TimeSeriesVisualization`) that are based on the JFreeChart library to create interactive time series charts.

Certainly, this list of technical visualization types is extensible, as new analyses may require individual visualization support. In the following, implemented analyses that make use of these visualizations to display their results are explicated with regards to their contents.

4.3.1 Class Dependency

The `ClassDependency` plugin determines the dependencies that exist between the implementation classes of a monitored application. The screenshot in Figure 4.6 illustrates how class dependencies can be clearly visualized in graph structures. The two depicted graphs are based on the same data, namely the class dependencies of the popular JPetStore reference application (only dependencies among classes of JPetStore's catalog component are included). The graphs only differ in their technical way of visualization: The graph in the left view part of Figure 4.6 is generated using the Graphviz library, while the graph in the right view part is visualized with the Eclipse Zest library. The latter allows for customizing the automatic graph layouts, as a user can interactively drag and drop the graph nodes to other

⁹ <http://www.eclipse.org/gef/>

4.3. Visualization of Analysis Results

positions. The coloring of the nodes distinguishes the tier structure of the JPetStore packages.

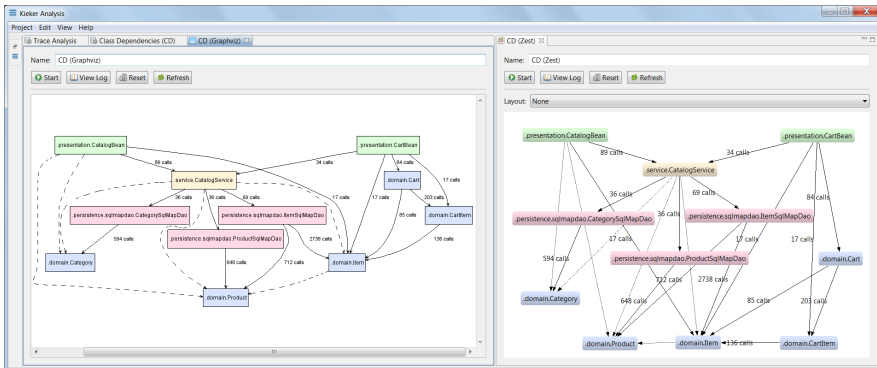


Figure 4.6. Visualization of effective class dependencies at runtime

The plugin extracts and weights dependencies that exist among the classes of an observed component. These reflect the effective coupling between classes at runtime and can complement information won by static code analysis. Each solid edge in Figure 4.6 is labeled with the number of calls having been observed between instances of the respective source and target classes. A call is either a method call, a constructor call, or a direct attribute access. Though static reverse engineering is able to locate code fragments where such calls are implemented, it is not possible to count the number of executions without running the system in a realistic usage scenario. Concerning this aspect, the plugin can enrich (reverse-engineered) design models by dynamic runtime information. In Figure 4.6, unweighted dependencies are represented by dashed edges and indicate that a class depends on another by using it as a method argument type, method return type, or attribute type. Other static dependencies such as inheritance, interface implementation, and method-local variables are not captured by the plugin. For more details on dependency models in object-oriented software systems, the reader is referred to [Systä et al., 2001; Streekmann and Hasselbring, 2008].

4. Continuous Software System Monitoring Integration

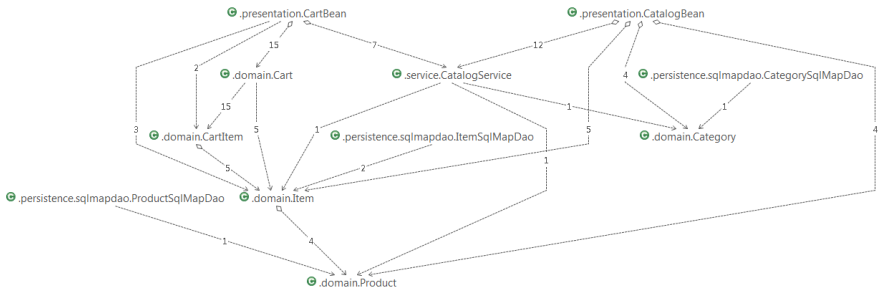


Figure 4.7. Class dependencies extracted by static code analysis

To compare the plugin's output with static code analysis results, Figure 4.7 depicts the class dependencies extracted with the reverse engineering tool STAN (Structure Analysis for Java)¹⁰. The graph in Figure 4.7 resembles the graphs in Figure 4.6, except for three differences. These regard to the edge weights and dependencies that do not exist at compile-time, but occur at runtime due to dynamic binding and framework delegation:

1. **Edge weights:** The quantified weights of the graph in Figure 4.7 indicate only how often a class is referenced from an dependent class, not how frequently a reference is affected at runtime
2. **Dynamic binding:** Code-level calls from `.service.CatalogService` to polymorphic interface methods are binded at runtime to `.persistence.*Dao` class methods. The interfaces are not shown in Figure 4.7 for clearness.
3. **Framework delegation:** Indirect calls from `.persistence.sqlmapdao.ItemSqlMapDao` to `.domain.Product` via the used persistence framework are not recognized by code reverse engineering.

¹⁰ <http://stan4j.com/>

4.3.2 Request Traces as UML Sequence Diagrams

The SequenceDiagram plugin visualizes the traces of different incoming system requests as UML sequence diagrams. The plugin receives requests from a TraceAnalysis filter. Requests with the same trace are aggregated. The observation frequency and the last observation time of each trace is reported. The number of possible unique traces is a priori unknown and unbounded. As the plugin runs during continuous operation, the traces are managed in a bounded collection from which older entries are removed if new traces arrive and the collection is already full. The collection size and removal discipline (first in first out, last recently used, last frequently used) is editable. The “Quick Sequence Diagram Editor” library¹¹ is applied to visualize the UML sequence diagrams.

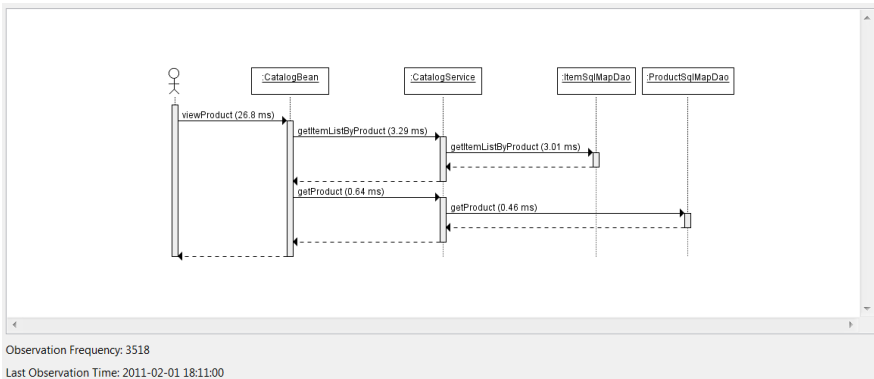


Figure 4.8. Request trace visualized as a UML sequence diagram

In comparison to sequence diagrams extracted by static code analysis, the plugin can attach information about the observed response time distribution to each synchronous method call of a trace. This is exemplified in Figure 4.8 by showing a trace excerpt of the JPetStore `.presentation.CatalogBean.viewProduct` service. To increase the conciseness

¹¹ <http://sdedit.sourceforge.net/>

4. Continuous Software System Monitoring Integration

of the sequence diagrams, method calls that do not exceed a minimum threshold concerning the required response time such as getters and setters can be hidden.

4.3.3 Analysis of Use Case Navigation Patterns

The UseCaseAnalysis plugin analyzes how users navigate through the system within a session. It tracks the session-internal transition probabilities and transition rates among the system use cases, i.e. in this case, the top-level services provided to the system users.

Figure 4.9 shows a transition graph capturing the four use cases of the JPetStore catalog domain. An editable parameter determines the stack level of each request trace from which the use cases are extracted. In the left view part of Figure 4.9, the number of processed sessions and requests is reported. Not all sessions are finalized, i.e. further incoming requests can be added to these sessions. A session timeout value (e.g. copied from the configuration of the application servers) has to be set that determines after which time without new requests a session is finalized. As the input order of requests from the trace analysis is not necessarily chronological, only finalized sessions are included in the displayed graph on the right.

Each directed edge of the graph is annotated with a tuple of its observed transition probability and transition rate. The outbound transition probabilities of each use case sum up to 1. Each node is annotated with a tuple of its relative steady state frequency and its observed mean rest time. The rest time is the time a user waits for and thinks about the response of a previous request until sending a new request to the system (response time + think time, see Figure 2.6). For each node, the sums of outbound and inbound transition rates are equal (local balance property). This is achieved by a closed workload scenario (when a session ends, a new session is created immediately) without significant queuing times (low resource utilization, think times with IS-like scheduling dominate response times). Assuming exponentially-distributed rest times, the transition graph can

4.3. Visualization of Analysis Results

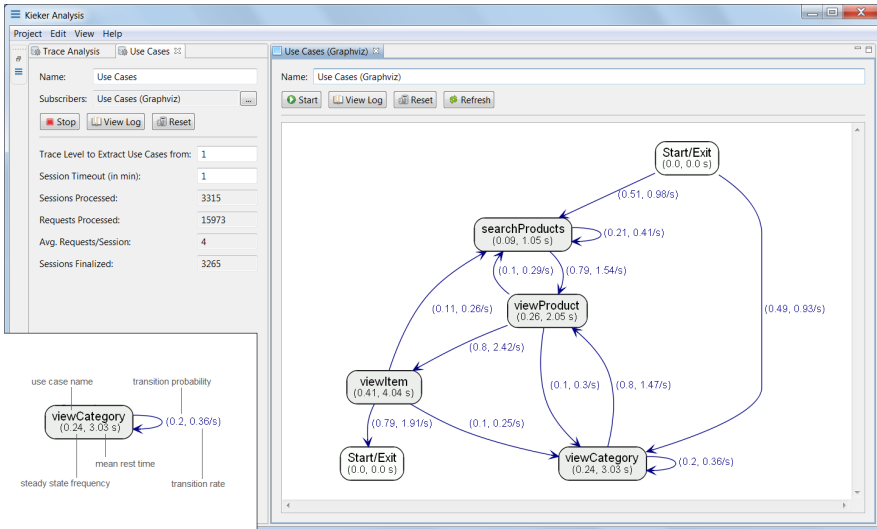


Figure 4.9. User profile visualization

be solved as a continuous-time Markov process. The intensity matrix Q is filled with transition rates, each being derived as a quotient of the corresponding transition probability and the use case rest time. When the number of active sessions remains constant, the resulting long-term steady state frequencies π^* indicate the proportion of users residing in each state. For example, in the scenario depicted in Figure 4.9, 24% of all users reside in the long term in the viewCategory use case, for which the mean rest time is 3.03 s. From this use case, 80% of the users request for viewing a product, while the other 20% view another category. The plugin utilizes the Apache Commons Math library¹² to solve the underlying equations (see Markov chains in Section 2.2.4: $\pi^*Q = \vec{0}$). More specific user behavior models with arbitrarily distributed rest times and session arrival rates (e.g. to account for seasonal variations and trends) have to be simulated.

¹² <http://commons.apache.org/math/>

4. Continuous Software System Monitoring Integration

4.3.4 Resource Utilization

The ResourceUtilization plugin processes and visualizes the utilization of hardware and software resources over time. If the analysis plugin is running, it iterates over all inbound log readers to start or to keep alive the remote ResourceUtilizationCollectors of the respective monitoring agents. Each monitoring agent provides resource utilization records for the system node/container it is deployed on. After the selection of a distinct monitoring agent, the plugin visualizes utilization time series of several resources observed by the selected monitoring agent. As illustrated by Figure 4.10, the provided time series include CPU utilization, used and total memory (OS and VM), as well as busy and available HTTP threads and JDBC connections. The time series are rendered by means of the JFreeGraph library. The axis scale of the displayed time series is editable.

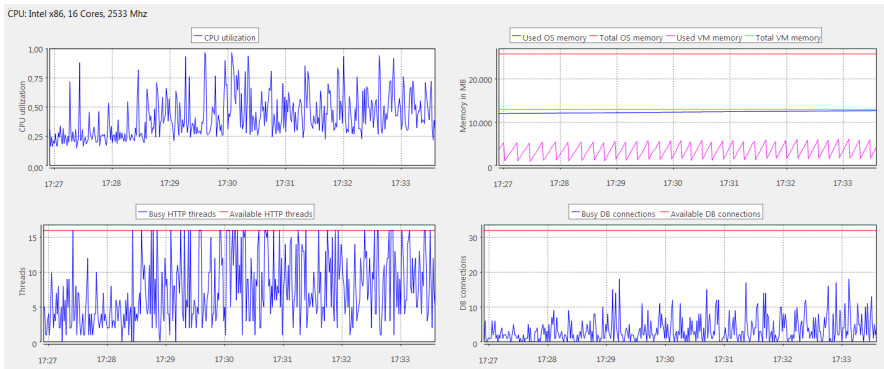


Figure 4.10. Resource utilization time series

4.3.5 Online Analytical Processing (OLAP) Integration

The OLAP plugin allows typical OLAP operations on multidimensional data sets like rollup, drill down, sclicing, dicing, and pivoting, as described in [Chaudhuri and Dayal, 1997]. The plugin receives Request instances

4.3. Visualization of Analysis Results

and adds them to a multidimensional data set managed by the Mondrian OLAP server¹³. The time dimension is continuously pruned concerning the available analysis granularity: While data from the recent day can be analyzed on a per-second level, data from the last month remains available only on an aggregated per-hour level, for example. Figure 4.11 shows a screenshot of the plugin using the Saiku OLAP frontend¹⁴ for data visualization. The screenshot shows the tabular result of a multidimensional query, where the average response times of several JPetStore operations (rows) are queried for each hour of the day (columns).

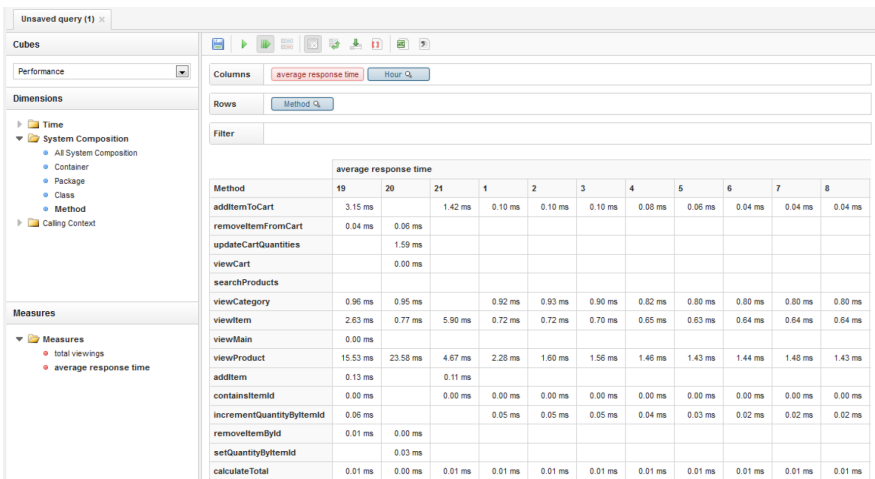


Figure 4.11. Online analytical processing (OLAP) integration

4.3.6 Performance Anomaly Analysis

The PerformanceAnomalyAnalysis plugin constructs a runtime model of the monitored software system in form of a calling context tree (CCT), as

¹³ <http://mondrian.pentaho.org/>

¹⁴ <http://www.analytical-labs.com/>

4. Continuous Software System Monitoring Integration

described in Section 3.2.5. Each calling context instance is associated with performance metrics (see Figure 4.12). The metrics include throughput measured by counting call frequencies, as well as summary statistics for the location (mean, median) and the dispersion (variance) of the observed response time distribution. Furthermore, anomaly scores are derived from the response time statistics, as described in Section 3.4.3. The CCT runtime model serves as input for the monitoring rules being presented in Section 3.3 and evaluated in Section 5.4.

The `PerformanceAnomalyAnalysis` plugin continuously receives `Request` instances from a `TraceAnalysis` filter to which it is subscribed. For each `Request`, it is iterated over its inherent tree structure of `OperationCalls` (see Figure 4.4). Each single `OperationCall` is processed: The corresponding `CallingContext` instance within the CCT, which aggregates operation calls with the same call stack over multiple requests, is looked up or created if it does not yet exist.

Each `CallingContext` is associated with a distinct operation of the monitored system. As illustrated in Figure 3.4, an `Operation` wrapper is interposed between a `CallingContext` and an `OperationSignature` of the system model to reference operation-level performance metrics. In this case, the referenced metric instances contain measures that are aggregated over all calling contexts in which the corresponding operation appears in the CCT. Furthermore, the operation wrapper stores if the underlying measuring point is recently monitored.

If monitoring of an operation is currently activated, each observed execution of this operation has to be processed to refresh the measures of the associated performance metrics. For that purpose, the implementation utilizes the complex event processing library `Esper`¹⁵ to employ a loosely coupled update mechanism for the performance metrics: For each operation execution, an event (class `OperationExecutionEvent`, containing information about the execution's calling context, its start time, and its duration) is sent to the `Esper` event processing runtime. Previously, during the initialization

¹⁵ <http://esper.codehaus.org/>

4.3. Visualization of Analysis Results

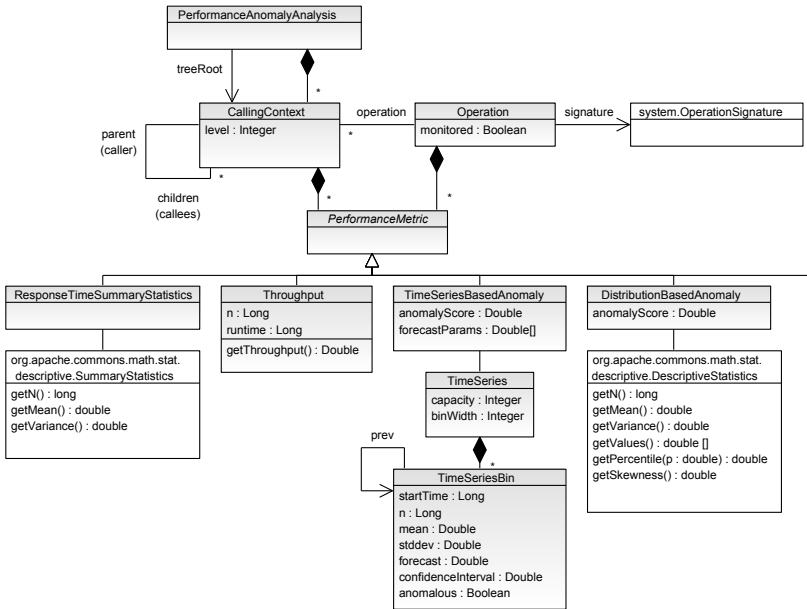


Figure 4.12. Performance anomaly analysis model

of the performance metrics, each metric registers a query statement with that runtime. Such a query expresses when and how a metric will be notified about new events, and particularly what kind of event information is passed to the metric’s update method. To illustrate the event-based metric update, the four metrics depicted in Figure 4.12 are described in more detail in the following:

- Throughput: An instance of the Throughput metric measures the total number of operation executions n_r for a particular context during analysis runtime r . Instead of updating the metric values with every single operation execution event, updates are made periodically after defined time batches, e.g. once per second. With each time batch, the metric is notified about the number of new operation executions n

4. Continuous Software System Monitoring Integration

and the start time of the latest execution t . The corresponding Esper query is:

```
select count(*) as n, max(startTime) as t
from OperationExecutionEvent(context='...').win:time_batch(1 sec)
```

The affinity to SQL is highly visible. By convention, Esper requires a subscriber for this query to implement a method with the signature `void update(Long n, Long t)`. The implementation of this update method increments n_r by n and refreshes the analysis runtime $r = \max(r, t - t_0)$, where t_0 is the analysis start time. Throughput is derived as n_r/r .

- `ResponseTimeSummaryStatistics`: This metric maintains summary statistics of the observed operation execution response times by means of the Apache Commons Math library. As it is not feasible for continuous operation to store all observed response time values, the implementation uses the `org.apache.commons.math.stat.descriptive.SummaryStatistics` data structure, which does not store single values in memory and thus is appropriate to compute statistics for large data streams. The summary statistics provide aggregated measures such as mean, sum, and variance that can be updated iteratively without high computation effort. Therefore, the response time x of each single operation execution is added to the summary statistics via an event subscriber method `void update(Double x)` invoked due to the registration of the following Esper query:

```
select duration as x from OperationExecutionEvent(context='...')
```

- `TimeSeriesBasedAnomaly`: This metric calculates an anomaly score based on time series analysis. The anomaly scoring procedure depends on several configuration properties, which are captured in the left view part of Figure 4.13 and referenced in the following by an id with the pattern `ts-id`. The metric manages an internal time series data structure, which contains a linked list of time series bins with a limited capacity (`ts-1`, see Figures 4.12 and 4.13). Each bin aggregates measures

4.3. Visualization of Analysis Results

for a sample of operation response times within a customizable time batch (*ts-2*). A new time series bin is created via an event subscriber method `void update(Long n, Double mean, Double stddev)` that is invoked each time the following Esper query outputs new batch data:

```
select count(*) as n, avg(duration) as mean, stddev(duration) as stddev
from OperationExecutionEvent(context='...').win:time_batch(1 sec)
```

If the capacity limit of the time series is reached, the oldest bin is removed before the new bin is inserted. Afterwards, the anomaly score is calculated, as described in detail in Section 3.4.3:

1. Forecast the expected response time μ_0 for the new bin based on a selected time series-based forecast model (*ts-6*), e.g. an ARIMA model.
- 2a. Calculate the test value t_0 based on the observed bin measures (n , mean \bar{x} , standard deviation s): $t_0 = (\bar{x} - \mu_0) \frac{\sqrt{n}}{s}$.
- 2b. Compare t_0 to the boundary of the confidence interval depending on a specified confidence level $1 - \alpha$ (*ts-3*): Is $|t_0| > t_{1-\alpha/2, n-1}$? If yes, the forecast value is outside the confidence interval and the recent bin sample is rated anomalous, i.e. $a_{\bar{x}} = 1$, otherwise not, i.e. $a_{\bar{x}} = 0$.
3. Update the overall metric's anomaly score a by exponential smoothing: $a_t = \delta a_{\bar{x}} + (1 - \delta) a_{t-1}$, where δ is *ts-4*.

Furthermore, the forecast model parameters are periodically updated (*ts-5*). For that, the time series of measured response times are forwarded from Java to the statistic tool R, in order to utilize the functions `HoltWinters` and `arima` in the R stats package¹⁶. These functions calculate the best-fitting forecast model parametrization for a given time series. The returned parameter values are used to forecast the expected response times.

- `DistributionBasedAnomaly`: This metric calculates an anomaly score based on clustering an observed distribution of response time samples, as described in Section 3.4.3. It maintains the

¹⁶ <http://cran.r-project.org/web/views/TimeSeries.html>

4. Continuous Software System Monitoring Integration

distribution of the incoming response time samples in form of a `org.apache.commons.math.stat.descriptive.DescriptiveStatistics` data structure. As this data structure stores each added response time value in memory, it internally implements a time-indexed FIFO queue with a fixed capacity limit (configuration property $d-2$ in Figure 4.13). The following Esper query is registered and periodically provides median response times, aggregated for customizable time batches ($d-1$).

```
select median(duration) as median
from OperationExecutionEvent(context='...').win:time_batch(1 sec)
```

For each time batch, the corresponding event subscriber method `void update(Double median)` is invoked. This method updates the metric's anomaly score as follows:

1. Add the new median value \tilde{x} to the distribution values (possibly causing the oldest value to be removed implicitly).
2. Run the LoOP algorithm by Kriegel et al. [2009] to rate the anomaly probabilities $\{a_{\tilde{x}}\}$ of the recent distribution values. The plugin does not reimplement the LoOP algorithm, but uses an existing implementation of the ELKI library¹⁷ [Achtert et al., 2010]. With this implementation, the anomaly probabilities for all distribution values are refreshed with each newly incoming median value. An incremental approach for data streams requiring less computational effort has been published by Pokrajac et al. [2007].
3. Update the metric's anomaly score a using a weighted moving average over the anomaly probabilities $\{a_{\tilde{x}}\}$, where newer anomaly values are weighted more heavily than older values.

Like the `ClassDependency` plugin, the `PerformanceAnomalyAnalysis` plugin uses the `Graphviz` library or the `Eclipse Zest` library to visualize a CCT. The right view part of Figure 4.13 shows such a CCT visualization for selected operations of the `JPetStore` catalog component. The user can interact with the visualization

¹⁷ <http://www.dbs.ifi.lmu.de/research/KDD/ELKI/>

4.3. Visualization of Analysis Results

- by clicking on an edge to toggle information about the frequency and the average response time of the underlying operation call (as shown for the CCT top level operations),
- by hovering over a node to popup a tooltip that provides further information about the operation’s signature and the calling context’s performance metrics (as shown for the operation `CatalogService.getItem`),
- by double-clicking a node to open a new view, which displays a time series of response times having recently been monitored in the selected calling context (see Figure 4.14),
- or by moving nodes to customize the tree layout.

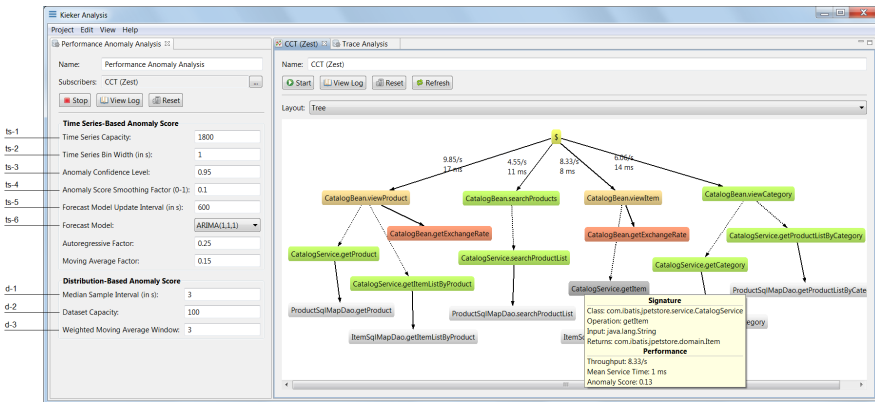


Figure 4.13. Interactive visualization of a calling context tree

The color of the operation nodes from green to red indicates their current anomaly score, based on one of the anomaly scoring procedures explicated above. Operations that are currently not monitored are colored in light gray. For instance in Figure 4.13, only operations with CCT level ≤ 2 are actually monitored. Among these, the operation `CatalogBean.getExchangeRate` appearing in two different calling contexts is rated anomalous. This anomaly is slightly propagated to upper levels of the CCT. Due to a possible decomposition of the monitored system into distributed components, requests may be traced across component borders. For the CCT in

4. Continuous Software System Monitoring Integration

Figure 4.13, remote calls are indicated by dashed edges, while solid edges indicate local calls.

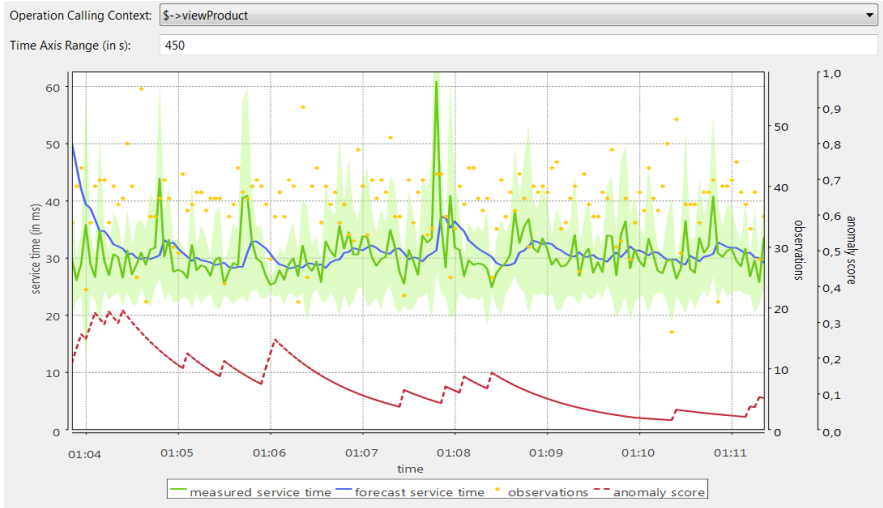


Figure 4.14. Performance time series visualization

The time series of monitored response times for a specific calling context are visualized by means of the JFreeChart library. Figure 4.14 exemplifies a response time curve of the `CatalogBean.viewProduct` operation (called from the CCT root level) for an interval of 450 s. The green line indicates the observed mean response time, surrounded by a light green confidence interval whose range depends on the observed variance and the specified confidence level. The blue line indicates the expected response time determined by the used forecast model. During failure-free operation, the forecast value is mostly within the confidence interval. By consequence, the anomaly score indicated by the red line does not rise considerably. Fault injection scenarios, in which the failure-free operation of the monitored software system is disrupted so that measurements and forecasts diverge, are evaluated in Section 5.1.

4.4 On-Demand Monitoring Adaptation

This subsection describes the design of the MonitoringAdaptation plugin, which has already been introduced in Section 3.2.7. In particular, the plugin enables a performance analyst to specify monitoring rules as presented in Section 3.3. The interior model of the MonitoringAdaptation plugin is depicted in Figure 4.15. As shown, an AdaptationAnalysis is a specific DataProcessingFilter which can be subscribed to other analysis filters such as a PerformanceAnomalyAnalysis instance. The plugin allows controlling the monitoring coverage of one or more MonitoringAgents that are part of the underlying analysis project. The total set of changeable probes and probe measuring points over all monitoring agents is queried and integrated via their remote interfaces (see Section 4.1). For each monitoring agent, known probes and probe measuring points can be activated and deactivated manually.

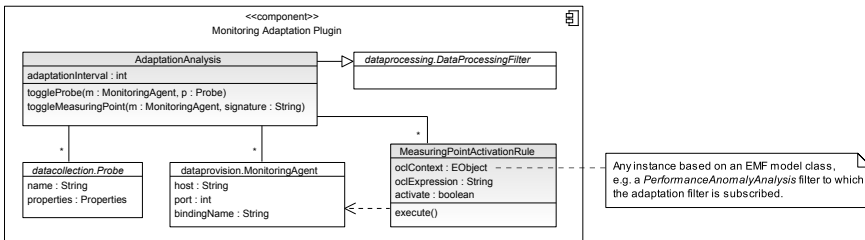


Figure 4.15. Model of the MonitoringAdaptation plugin

The corresponding dialog (indicated as ③), which summarizes the current monitoring status and allows toggling of probes and measuring points, is displayed at the bottom right of Figure 4.16: The dialog screenshot shows 3 probes spread over 2 monitoring agents named after their hosts *blade1-2*. The RequestEntryProbe and the OperationCallTracingProbe are activated on both agents. The ResourceUtilizationProbe is deactivated at *blade1*, while it is unknown at *blade2*, i.e. not instrumented. In addition to the activation status, further probe-specific properties are editable at runtime

4. Continuous Software System Monitoring Integration

as well, e.g. the collection interval of a ResourceUtilizationProbe controlling in which frequency new utilization data is gathered (as depicted in the left dialog ① of Figure 4.16).

The operation-level measuring points are displayed in a tree structure according to their package and class membership. For instance, monitoring is currently deactivated for all operations within the package `com.ibatis.jpetstore.persistence`. Regarding the class `CatalogBean`, monitoring is activated for all of its operations at `blade1`, while at `blade2` only selected operations are monitored. Furthermore, it is notable that the classes `AccountBean` and `OrderBean` are completely unknown at `blade1`, i.e. either the system component which these classes belong to has not been deployed at this system node, or none of their class methods has been instrumented and executed at least once.

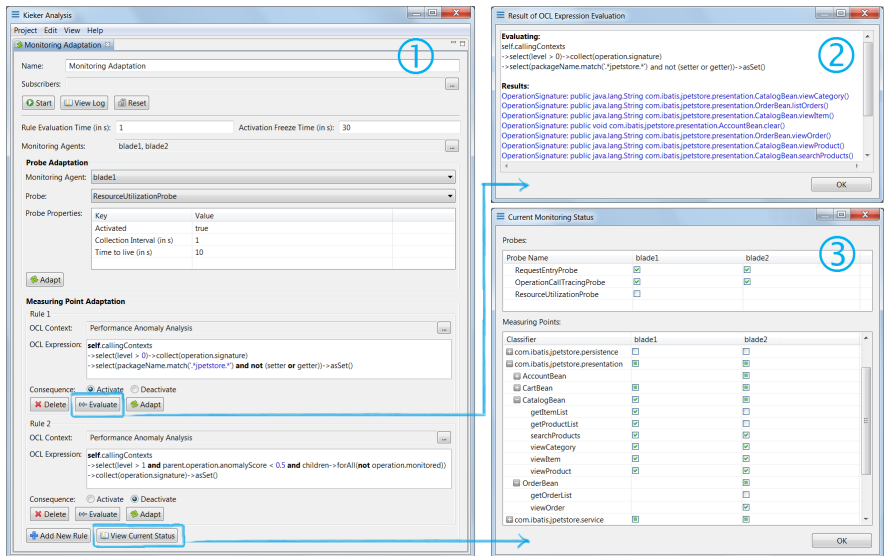


Figure 4.16. Kieker plugin for runtime monitoring adaptation: ① specification of OCL-based monitoring rules, ② monitoring rule evaluation result, ③ manual probe and measuring point activation

4.4. On-Demand Monitoring Adaptation

Besides the possibility of manual measuring point (de)activation, the plugin allows the specification of OCL-based monitoring rules. The continuous evaluation and application of these rules realizes the self-adaptive monitoring approach to detect performance anomalies, as proposed in Section 3.4. As depicted in Figure 4.15, each `MeasuringPointActivationRule` has an OCL context element, an OCL expression that selects a set of operation signatures and a flag indicating whether the rule's consequence should be to activate and to deactivate monitoring of the selected operations. Concerning its implementation, the `MonitoringAdaptation` plugin depends on and uses libraries of two sub-projects of the Eclipse Modeling Project named `EMF Query2` and `OCL`¹⁸, which are provided in form of Eclipse plugins (versions from the Eclipse Indigo release 2011). Using these Eclipse plugins, it is possible to evaluate arbitrary valid OCL expressions on EMF model instances, i.e. any `EObject` instance can serve as the OCL context element referenced by the keyword `self`. This allows querying data from EMF-based runtime models by means of OCL. The queries are type safe, i.e. the `Query2` plugin contains a type checker which checks the query expressions against the EMF model. Alternatively, `Query2` supports expressing the queries using a SQL-like syntax based on `Xtext` or using an object-oriented Java API¹⁹. By convention, the query expression should deliver a collection of `OperationSignatures` as its result, which is delegated to the remote adaptation interface of the monitoring agents (see Section 4.1) for measuring point (de)activation.

When the `AdaptationAnalysis` thread is started, it successively executes all created rules in the configured order. The rule order is important if multiple rules affect the same measuring points, but have different consequences (activation vs. deactivation). In this case, a posterior rule will override the results of a prior rule. The execution of all rules is repeated periodically depending on the configured adaptation interval value.

¹⁸ <http://www.eclipse.org/modeling/mdt/?project=ocl>

¹⁹ <http://wiki.eclipse.org/EMF/Query2>

4. Continuous Software System Monitoring Integration

The left dialog ① in Figure 4.16 shows two configured monitoring rules:

Monitoring rule R_3 : This rule activates monitoring for all operations with a package name that contains the pattern “jpetstore”, excluding simple getter or setter methods. OCL expression:

```
self.callingContexts→select(level > 0)→collect(operation.signature)
→select(packageName.match('.*jpetstore.*') and not (setter or getter))→asSet()
```

Monitoring rule R_4 : The second rule is a negation of the rule R_2 explicated in Section 3.3. It collects and deactivates monitoring for all operations whose caller is not rated anomalous (`parent.operation.anomalyScore < 0.5`) and for which none of its callees is currently monitored (`children→forAll(not operation.monitored)`). Operations that are part of the external system interface (CCT level = 1) are excluded from this self-adaptive deactivation. OCL expression:

```
self.callingContexts→select(level > 1 and
parent.operation.anomalyScore < 0.5 and children→forAll(not operation.monitored))
→collect(operation.signature)→asSet()
```

The result of R_3 is unchanging, as long as no new operations are added to the concerned packages, e.g. by deployment of system patches or new releases. The result of R_4 varies continuously, as it references performance metrics such as the anomaly score that change their values at runtime. It is notable that R_4 does not reference the performance metrics of a possibly deactivated measuring point itself, but those of related measuring points, i.e. the operation caller (`parent.operation...`) in this case. The reason is that performance metrics of an unmonitored operation will not change unless new records arrive via its interception measuring point. Thus, it is useful to setup an additional rule, which activates a periodically changing random selection of measuring points. By this means, sparsely distributed sample observations can be made to check over the learned expected behavior.

Monitoring rule R_5 : The following rule selects 10 random operations having a measured mean response time > 20 ms. OCL expression:

4.4. On-Demand Monitoring Adaptation

```
self.callingContexts→select(level > 0 and responseTimeSummaryStatistics.mean > 20)  
→collect(operation.signature)→random(10)
```

The used random-function (as well as the match-function in R_3 above) are not part of the OCL standard, but are introduced as customizations by the Kieker MonitoringAdaptation plugin. The Eclipse OCL project supports customizing the Ecore-based OCL environment to define such additional custom functions²⁰. The custom functions are integrated into the rule editor of the MonitoringAdaptation plugin, which provides code assist and syntax highlighting to facilitate the rule specification. Each rule can easily be tested before it is activated for continuous operation. The top right dialog ② of Figure 4.16 illustrates the result of a rule's test evaluation. The result consists of a collection of OperationSignatures.

²⁰ How to customize the OCL environment: <http://help.eclipse.org/indigo/?topic=/org.eclipse.ocl.doc/references/overview/advanced/customization.html>

Experimental Evaluation

This chapter presents the conducted experimental evaluations to demonstrate the feasibility and practicability of the adaptive monitoring approach. Section 5.1 evaluates the time series-based forecast models, as introduced in Section 3.4.3. Afterwards, the accuracy of the proposed anomaly scoring approaches is compared in Section 5.2. Section 5.3 studies the computational overhead coming along with monitoring and analysis of a system's runtime behavior. An integrated evaluation of the rule-based self-adaptability concerning the monitoring coverage is provided in Section 5.4. Accordingly, each section addresses one of the research questions listed in Section 1.2:

- RQ #9 (Section 5.3): How much overhead is caused by extensive software system monitoring?
- RQ #10 (Section 5.1): To which degree of accuracy is it possible to forecast response times of software services?
- RQ #11 (Section 5.2): How effective are the proposed anomaly scoring procedures in comparison?
- RQ #12 (Section 5.4): Is a self-adaptive control of the monitoring coverage feasible in practice?

Following the goal/question/metric paradigm of Basili [1992], the corresponding metrics, which are used to address the goals by quantified experiments results, will be introduced in the following sections themselves. For each section, the experiment setup is described before the experiment results are presented and valued.

5. Experimental Evaluation

5.1 Time Series-Based Forecast Model Evaluation

The goal of this experiment is to evaluate the quality of response time forecasts based on time series analysis. The research question to be studied is: "To which degree of accuracy is it possible to forecast response times of software services?" The metric used to quantify the forecast accuracy is the mean absolute percentage error (MAPE) considering a time series of forecast values $\{f_1, \dots, f_T\}$ and a corresponding time series of actual posterior measurement values $\{m_1, \dots, m_T\}$:

$$\text{MAPE} = \frac{1}{T} \sum_{t=1}^T \left| \frac{m_t - f_t}{m_t} \right|$$

MAPE values are computed and compared for the different forecast models presented in Section 3.4.3. The forecast models are evaluated in two experiment lab scenarios, each covering multiple software services. The experiment setup of both scenarios is described in the following.

Experiment setup: In Scenario F_1 , the JPetStore reference application is instrumented and observed as the system under test. In Scenario F_2 , the timing behavior of the SPECjEnterprise2010 industry standard benchmark is studied. The benchmark is aimed at performance measurement and characterization of Java EE 5 servers and supporting infrastructure including JVM, database, CPUs, disks and network.

Deployment diagrams of the systems under test and the related workload drivers are provided by Figure 5.1 for Scenario F_1 and by Figure 5.2 for Scenario F_2 . The hardware resources used in the experiments are four identical interconnected servers named *blade<1-4>*, each with the following setup: Sun Blade X6270 with 2x Intel Xeon E5540, total 8 cores at 2.53 GHz, 24 GB RAM, ZFS RAID, SunOS 5.1, Java HotSpot x86 Server VM 1.6.

5.1. Time Series-Based Forecast Model Evaluation

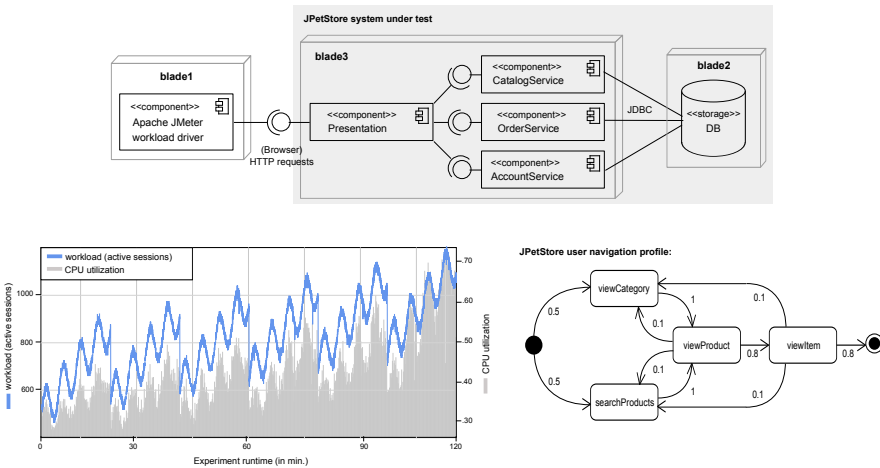


Figure 5.1. Forecast model evaluation – JPetStore scenario setup: deployment view (top), varying workload and resource utilization (bottom left), usage profile (bottom right)

In Scenario F_1 , the monitored system is stressed with custom intensity-varying workload. Apache JMeter including the Markov4JMeter extension¹ [van Hoorn et al., 2008] is used as a workload driver tool. The number of concurrent users is constructed to provoke a close-to-reality workload with seasonal variation and trend. The workload curve is depicted in correlation with the resulting total CPU utilization in the bottom left of Figure 5.1. The CPU has been the major resource bottleneck to impact varying response times. The underlying user behavior including four selected services of the JPetStore application is shown in the bottom right of Figure 5.1. The edge directions and weights indicate inner-session transition probabilities between the services for which response times are measured and forecasted.

With Scenario F_2 , the user behavior and the workload is controlled by the SPECjEnterprise2010 benchmark itself. The benchmark uses the Faban

¹ <http://markov4jmeter.sourceforge.net/>

5. Experimental Evaluation

harness and benchmark tool² to generate workload as a function of a transaction rate parameter. This transaction rate has to remain constant during runtime when the benchmark results are intended to be officially published. As it is more important for this experiment to simulate varying workload, the transaction rate fluctuates as shown in the right of Figure 5.2.

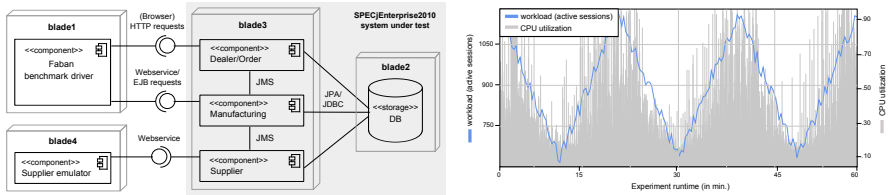


Figure 5.2. Forecast model evaluation – SPECjEnterprise2010 scenario setup: deployment view (left), varying workload and resource utilization (right)

In both scenarios, the measured response times are processed by Kieker’s PerformanceAnomalyAnalysis plugin to compute the forecast response times. The experiment setup contains two variables: (1) the forecast model update interval, which specifies how frequently the forecast model parameters are recomputed, and (2) the time series bin width, which specifies how frequently new response time samples are taken (cf. Section 4.3.6). Both experiment scenarios have been replicated with different variable combinations: namely the Cartesian product of the forecast model update intervals $\{10, 15, 20, 30\}$ (in min.) and the time series bin widths $\{3, 4, 5, 8\}$ (in s).

Experiment results: Table 5.1 summarizes the results comparing the mean forecast errors of the different forecast models. For each forecast model, the table reports the MAPE values aggregating all samples of the experiment run and averaged for all variable combinations of forecast model update intervals and time series bin widths. Besides the forecast models explicated in Section 3.4.3, results are given and compared to more pragmatic forecast approaches x_t (forecast is latest observed value) and \bar{x}_t (forecast is average

² <http://java.net/projects/faban/>

5.1. Time Series-Based Forecast Model Evaluation

of preceding observations). The results are listed line-by-line for each monitored service of the particular system under test.³

Experiment scenario	Evaluated software service	Mean absolute percentage error (MAPE) of forecast model					
		SES	DES	ARIMA(1,0,1)	ARIMA(1,1,1)	x_t	\bar{x}_t
F ₁	searchProducts	13,1%	14,3%	12,9%	13,9%	15,9%	19,2%
	viewCategory	14,4%	15,9%	14,4%	14,6%	17,6%	20,1%
	viewProduct	12,3%	13,9%	12,2%	12,4%	15,2%	19,3%
	viewItem	13,1%	14,4%	12,7%	13,8%	15,7%	19,7%
	scenario average	13,2%	14,6%	13,0%	13,7%	16,1%	19,6%
F ₂	doInventory	23,4%	20,8%	22,6%	20,1%	22,6%	57,4%
	doVehicleQuotes	22,2%	20,7%	20,4%	20,8%	22,8%	57,7%
	doRemoveFromShoppingCart	21,5%	17,6%	17,6%	14,4%	21,1%	33,8%
	doShoppingCart	21,0%	21,0%	21,6%	19,3%	22,8%	47,3%
	doHome	21,4%	21,6%	27,1%	18,3%	22,6%	57,3%
	doDeferredPurchase	22,3%	24,9%	20,5%	10,9%	22,3%	47,7%
	doAddToShoppingCart	20,9%	21,7%	21,9%	21,6%	24,3%	53,7%
	doClearCart	22,3%	19,2%	23,0%	19,1%	24,0%	47,1%
	scenario average	21,9%	20,9%	21,8%	18,0%	22,8%	50,3%

Table 5.1. Forecast model evaluation – mean forecast errors

Regarding F₁, ARIMA(1,0,1) and SES provide the best forecast results with comparatively low MAPE values of about 13%. The simple forecasts x_t and \bar{x}_t achieve the worst results as they do not cope with the intensity-varying workload (trend and seasonal variations). In case of scenario F₂, the forecasts are less precise over all forecast models with MAPE values of about 20–22%. This is caused by the characteristics of the SPECjEnterprise2010 benchmark aiming at heavy resource utilization with temporary capacity overload (provoked by sending thousands of small requests/s). Scenario F₂ differs from F₁ in respect to the major resource bottleneck: While the CPU with approx. processor sharing (PS) scheduling has been the bottleneck in F₁, it

³ The design document of the SPECjEnterprise2010 benchmark describes its provided services:

<http://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html>

5. Experimental Evaluation

is database I/O with FCFS scheduling in F_2 . The latter causes much more scattered outliers, which the forecast models have to cope with.

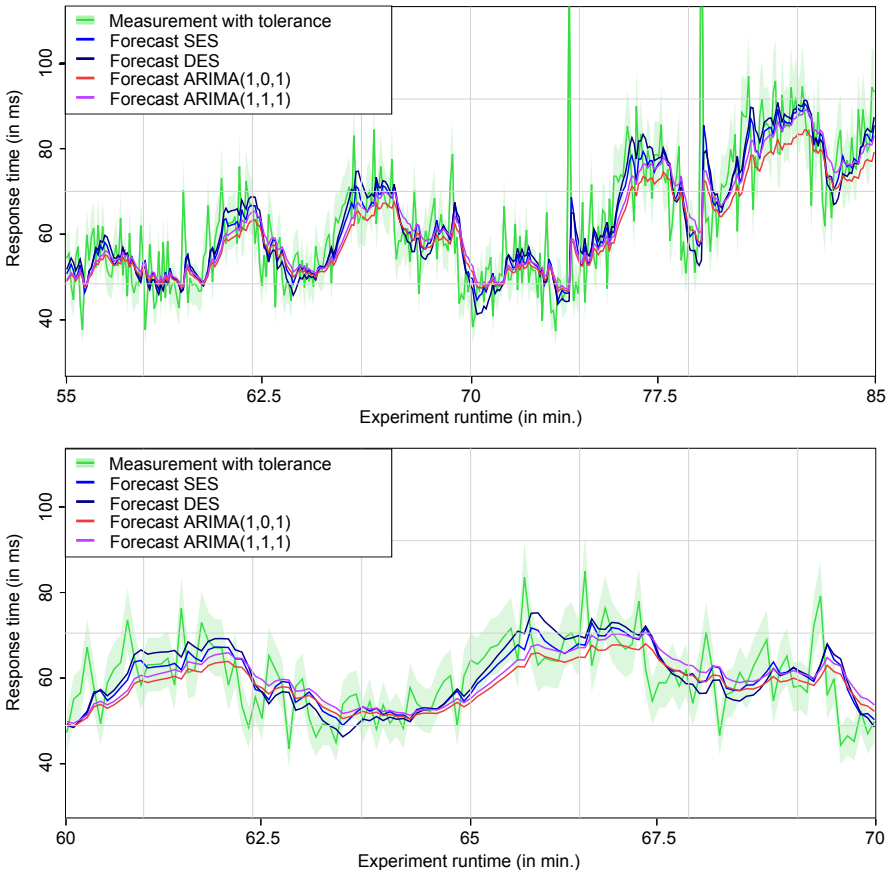


Figure 5.3. Line charts of measured and forecasted response times

In both scenarios, the forecast error is significantly negatively correlated (Pearson correlation coefficient $r \approx -65\%$) with the time series bin size, as a greater bin and sample size leads to outlier smoothing, and by consequence to a smaller forecast error. However, greater bin sizes reduce the ability

5.1. Time Series-Based Forecast Model Evaluation

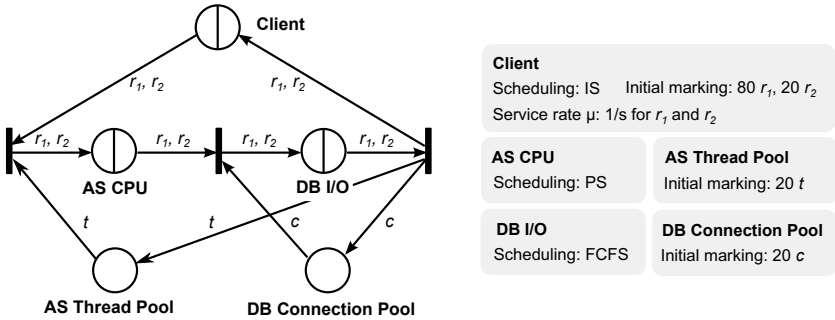
of undelayed reaction on arising anomalies. The forecast model update interval did not have a significant impact on the results.

Figure 5.3 visualizes the predicted response time values of the evaluated forecast models in comparison to each other and to the measured response time. The line charts display an excerpt of an experiment run concerning the JPetStore `viewProduct` service. While the upper chart provides an overview of 30 min., the lower chart presents a more detailed view on 10 min. Both charts indicate how the forecast models try to anticipate and to adapt to the measured time series.

Limitations of the time series-based forecast models: As stated above, response time forecasting based on time series analysis provides better results, i.e. less forecast errors, if the monitored system's resource bottleneck uses PS scheduling instead of FCFS. This effect occurs particularly if multiple request types with varying resource demands are served in parallel and thus compete for the same resources, e.g. disk I/O. The resulting limitation of time series-based forecast approach is made clear by a simulation of a simplified software system: Figure 5.4 illustrates a QPN (cf. Section 2.2.4) where two request types r_1, r_2 compete for two active resources (AS CPU, DB I/O) while being processed. A passive resource is required before an active resource can be used, i.e. a thread t for the application server (AS) and a connection handle c for the database (DB).

This simple system has been simulated with different request services rates using the QPME tool. The simulation results are listed at the bottom of Figure 5.4 and show how the bottleneck's scheduling discipline impacts the request service times: The request service rates are set to 0.5/ms so that a request ideally needs 2 ms at each active resource to be served. The results of the first simulation run with no bottleneck report that in fact the mean service time is about 2.5 ms due to concurrent resource usage. In the following simulation runs, the service rate for r_2 at AS CPU and DB I/O respectively is increased. While a PS bottleneck does not cause a shift in the proportion of service times (r_1 : 4 ms, r_2 : 40 ms), a FCFS bottleneck favors requests with relatively greater service times such as r_2

5. Experimental Evaluation



Simulation run (bottleneck)	Settings: Service rates μ (/ms)				Results: Avg. service times (ms)			
	AS CPU		DB I/O		AS CPU		DB I/O	
	r_1	r_2	r_1	r_2	r_1	r_2	r_1	r_2
1 (none)	0,5	0,5	0,5	0,5	2,483	2,486	2,49	2,488
2 (AS CPU: PS)	0,5	0,05	0,5	0,5	4,222	40,371	2,484	2,486
3 (DB I/O: FCFS)	0,5	0,5	0,5	0,05	2,483	2,486	17,873	35,254

Figure 5.4. QPN: System bottleneck's scheduling impacts service time predictability

(35 ms) and leads to strong outliers for the competing requests such as r_1 (18 ms). In case of a FCFS bottleneck and multiple request types with considerable differences in their resource demands, the response times will be scattered and thus more difficult to predict based on a historic time series due to the large standard deviation. That is why the forecast errors of the SPECjEnterprise2010 scenario F_2 are higher than those of the JPetStore scenario F_1 .

5.2 Anomaly Scoring Evaluation

This experiment aims at evaluating the time series-based and distribution-based anomaly scoring procedures, as described in Section 3.4.3.

5.2. Anomaly Scoring Evaluation

It follows the research question: “How effective are the proposed anomaly scoring procedures in comparison?” The metrics that are applied to address this question are commonly used classifier quality criteria, including recall, specificity, precision, and derived F-scores [Salfner et al., 2010]. The recall r (or sensitivity) measures the true positive rate, i.e. the proportion of response time samples that are rightly classified as anomalous (true positives t_p) of all samples that actually are positive ($t_p +$ false negatives f_n). The specificity s measures the true negative rate, i.e. the proportion of samples that are rightly classified as not anomalous (true negatives t_n) of all samples that actually are negative ($t_n +$ false positives f_p). In practice, there is a trade-off between recall and specificity as both measures are usually inversely proportional to each other.

$$r = \frac{t_p}{t_p + f_n}, \quad s = \frac{t_n}{t_n + f_p}, \quad p = \frac{t_p}{t_p + f_p}, \quad F_\alpha = \frac{(1 + \alpha^2) \cdot p \cdot r}{\alpha^2 \cdot p + r}$$

The precision p measures the positive predictive value, i.e. the proportion of samples that are rightly classified as positives of all positively classified samples. The F_α -score provides a metric combining recall and precision, where the parameter α determines the weighting. In the results, the F_1 - and $F_{0.5}$ -score are reported. The F_1 -score weights recall and precision equally. Concerning the efficacy of an anomaly detection approach for a continuously running system, the number of false positives (false alarm) has to be kept small. Thus, the $F_{0.5}$ -score that weights precision stronger than recall is added to the results report.

Experiment setup: To evaluate the effectiveness of the continuously updated operation-level anomaly scores, a supervised validation is required. For a given time series of operation response time samples, each observed sample has to be associated with a “correct” target value indicating whether the sample should be rated anomalous or not. A challenge ahead is to specify these correct target values.

5. Experimental Evaluation

A first idea is to construct design-oriented performance models (e.g. by means of UML-SPT or PCM as described in Section 2.2.4) that allow annotating operations with required, predicted, and measured timing behavior. In this case, significant deviances between required and measured response times could indicate anomalies. In practice, SLOs containing the required response times do usually not yet exist on a fine-grained level of distinct software operations. Besides, typical SLOs, which require the major proportion of all requests (e.g. per day) to respond below a specified time threshold, are not sufficient to judge on single response time samples being collected in a time interval of a few seconds.

Thus, a more pragmatic way is taken: As part of the experiment setup, anomalies are injected manually on purpose at specific time segments. All response time samples that are taken from such an outlier time segment are indicated with a target anomaly score of 1. For all other samples this target value is 0. As the actual anomaly score is no binary classifier, a custom true/false positive/negative assignment, corresponding to Listing 5.1, is used for the samples' validation.

```
for (ResponseTimeSample s : getSamples()) { // iterate over all samples of the experiment run
    double as = s.getAnomalyScore(); // actual anomaly score after processing the current sample
    if (s.getTargetValue() == 1) { // case of anomalous time segment
        t_p += min(2 * as, 1); // increase true positives
        f_n += 1 - min(2 * as, 1); // increase false negatives
    }
    else { // case of not anomalous time segment
        t_n += as < 0.5 ? 1 : 0; // increase true negatives
        f_p += as > 0.5 ? 1 : 0; // increase false positives
    }
}
```

Listing 5.1. Anomaly score validation – true/false positives/negatives

As discussed in Section 2.2.2 and summarized in Figure 2.8, there are different ways to disturb the timing behavior of software components. This experiment is subdivided into two scenarios using different software services under test and a different way to cause the necessary performance anomalies.

5.2. Anomaly Scoring Evaluation

Scenario AS₁: In this scenario, a synthetic software service is studied that depends on an external service with instant and unexpected changes in its timing behavior. These changes are to be detected as anomalies. During the experiment run, the response times of the observed service increase by 25%, 50%, and 100%, each for a time segment of 30 s, and decrease by 50% for 60 s. Figure 5.6 illustrates the response time curve of this scenario and highlights the time segments where critical anomaly score peaks are expected.

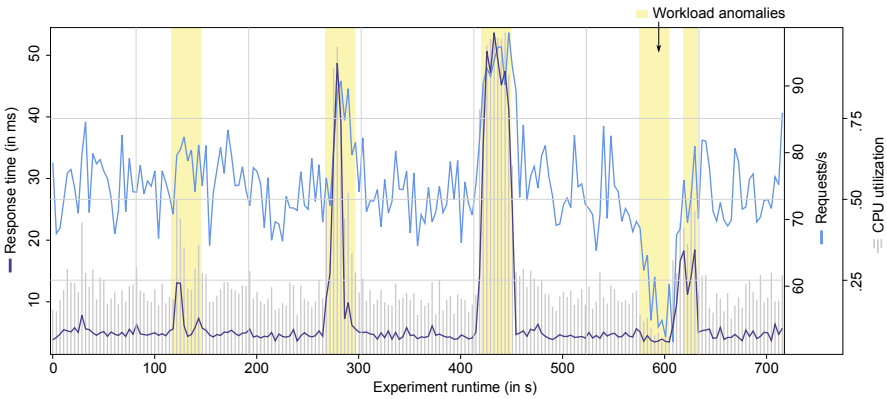


Figure 5.5. Anomaly score evaluation – workload setup

Scenario AS₂: In this scenario, the software services of the SPECjEnterprise2010 benchmark are studied to test the anomaly scoring procedures with an advanced Java EE-based software system. By default, the workload driver that comes along with the SPECjEnterprise2010 benchmark generates workload that does not vary in its intensity. For this scenario, this constant usage profile is interrupted by abrupt increases or decreases of the workload for short time segments to provoke performance anomalies. Corresponding to Scenario AS₁, the workload intensity is increased by 10%, 25%, and 50%, each for a time segment of 30 s, and decreased by 50% for 60 s. These time segments of higher or lower workload are to be detected as anomalies. Figure 5.5 illustrates how the changes in the usage profile indirectly affect the response time of the SPECjEnterprise2010 services. The

5. Experimental Evaluation

depicted workload, utilization, and response time curves are averaged over all monitored services listed in Table 5.2.

For both scenarios, the anomaly scoring procedures have been parametrized the same as follows (see Section 4.3.6 for the parameters' description):

Time series-based anomaly scoring:

- Time series bin width: 1 s
- Time series capacity: 300 ($\hat{=}$ 5 min. history)
- Anomaly confidence level: 0.99
- Forecast model: SES (fixed smoothing factor: 0.1)
- Anomaly score smoothing factor: 0.2

Distribution-based anomaly scoring:

- Median sample interval: 1 s
- Dataset capacity: 120 ($\hat{=}$ 2 min. history)
- Moving average window: 3

Experiment results: The above parameter values have been selected due to previous explorative testing. Unfortunately, there is no universal ideal parameter setting for all software systems to be monitored. Instead, it is required to adjust the parameter values to the response time dispersion of the monitored services in individual cases. In doing so, the objectives and the weighting concerning recall, specificity, and precision of the involved performance analysts are particularly relevant.

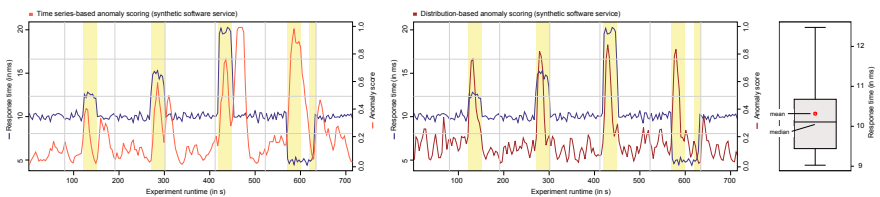


Figure 5.6. Anomaly score evaluation – results visualization (Scenario AS₁)

Scenario AS₁: Figure 5.6 illustrates how the time series-based anomaly score (abbr. TS-AS, orange curve in left line chart) and the distribution-based

5.2. Anomaly Scoring Evaluation

Experiment scenario	Evaluated software service	Time series-based anomaly score					Distribution-based anomaly score				
		recall	specificity	precision	F ₁	F _{0.5}	recall	specificity	precision	F ₁	F _{0.5}
AS ₁	<i>synthetic service</i>	0.58	0.95	0.85	0.69	0.78	0.88	0.99	0.92	0.9	0.91
AS ₂	doSell	0.67	0.96	0.86	0.76	0.82	0.7	0.98	0.83	0.76	0.8
	doInventory	0.79	0.96	0.88	0.83	0.86	0.72	0.98	0.78	0.75	0.77
	completeWorkOrder	0.48	0.98	0.91	0.63	0.77	0.62	0.99	0.9	0.74	0.83
	doVehicleQuotes	0.95	0.56	0.42	0.58	0.47	0.71	0.98	0.82	0.76	0.8
	doPurchase	0.33	1.0	1.0	0.5	0.71	0.61	0.99	0.9	0.73	0.82
	doShoppingCart	0.52	1.0	1.0	0.68	0.84	0.66	0.99	0.84	0.74	0.8
	doHome	0.74	0.92	0.76	0.75	0.75	0.74	0.98	0.79	0.76	0.78
	doDeferredPurchase	0.46	1.0	1.0	0.63	0.81	0.7	0.98	0.83	0.76	0.8
	doAddToShoppingCart	0.79	0.97	0.89	0.83	0.87	0.65	0.98	0.8	0.72	0.76
	doCancelOrder	0.52	0.98	0.9	0.66	0.79	0.66	0.98	0.8	0.72	0.77
average of above	0.63	0.93	0.86	0.69	0.77	0.68	0.98	0.83	0.74	0.79	

Table 5.2. Anomaly score evaluation – results table

anomaly score (abbr. D-AS, red curve in right line chart) rise in the highlighted anomalous time segments. Having risen in reaction to an abrupt response time ascent, both anomaly scores quickly decline afterwards. The reason is that in this scenario the response times within an anomalous time segment are very homogeneous and thus are quickly understood as the new expected timing behavior. In comparison to the D-AS, the TS-AS rises to a second peak in the end of each anomalous time segment when the response times drop back to the normal level. The D-AS does not show this reaction as its dataset capacity (120 s) is considerably longer than the anomalies last (30–60 s). By consequence, the previous normal data is still part of the dataset submitted to the LoOP algorithm, which does not consider the chronological order of observations.

Looking at the measures in Table 5.2 for Scenario AS₁, the recall of the TS-AS is comparatively low as the TS-AS reacts slightly delayed and does not go up sufficiently during the first anomalous time segment (where it remains arguable if this time segment actually is anomalous). Concerning Scenario AS₁, the other primary measures being in the range of 0.85–1.0 provide experimental validation of the anomaly scoring procedures' effectiveness.

5. Experimental Evaluation

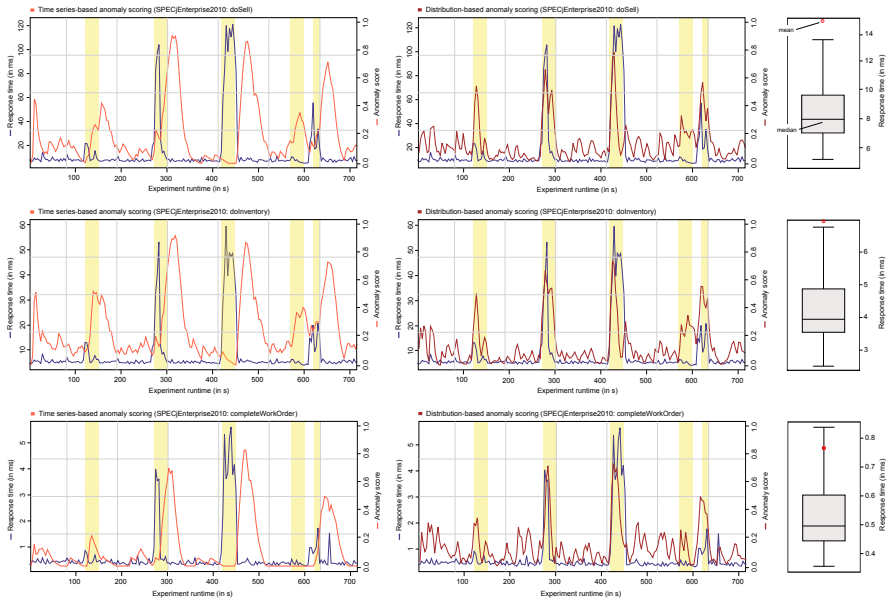


Figure 5.7. Anomaly score evaluation – results visualization (Scenario AS₂)

Scenario AS₂: The measures for the monitored services of the SPECjEnterprise2010 are listed in Table 5.2. Figure 5.7 shows the anomaly score curves over experiment runtime for three selected services: doSell, doInventory, and completeWorkOrder. Additionally, response time boxplots for each these services are shown on the right of Figure 5.7. The boxplots illustrate the typical right-skewed response time distribution of software services. The mean response times are significantly larger than the median values due to extreme outliers in the anomalous time segments. For this reason, the upper whiskers (default position at the 1.5 interquartile range of the 3rd quartile) are remarkably low, and the response time axes scales of the boxplots and the line charts differ heavily.

It is clearly visible that the TS-AS reacts delayed due to a low SES smoothing factor. This delay has been compensated when calculating the measures for

5.3. Monitoring and Analysis Cost Evaluation

Scenario AS₂ in Table 5.2. Nevertheless, the D-AS tends to have a better recall than the TS-AS. Both, anomaly scores provide a good specificity and precision, and thus reach acceptable F-scores in a range of 0.69–0.79. While the D-AS has a better F₁-score, the TS-AS is better regarding the F_{0.5}-score.

5.3 Monitoring and Analysis Cost Evaluation

Monitoring biases the runtime performance of the observed software system. The monitoring overhead produced at the server resources hosting the operational software system has to be kept deliberately small. Thus, the monitoring agents require an efficient implementation limited to necessary computational aspects. As the centralized analysis of the monitoring data is hosted on a dedicated server resource, its computational cost is comparatively not that critical.

This section contains a quantitative evaluation of the computational cost on monitoring and analysis side, related to the Kieker monitoring framework. The studied research question is: "How much overhead is caused by extensive software system monitoring?"

5.3.1 Monitoring Cost

The Kieker Monitoring component has been employed in the productive systems of a telecommunication company [van Hoorn et al., 2009b] and a digital photo service provider [Rohr et al., 2010]. These previous case studies confirmed its practicability and robustness. Regarding the monitoring cost, the industrial partners were not able to perceive any monitoring overhead due to the instrumentation of the monitoring probes. Thus, lab experiments have been conducted following the goal to quantify and to decompose the monitoring overhead. The overhead of AOP-based interception has already been studied in related work regarding outdated AspectJ predecessor versions [Greenwood and Blair, 2006].

5. Experimental Evaluation

In Figure 5.8, monitoring costs are apportioned for instrumentation (Δ_I), data collection (Δ_C), and data logging (Δ_L). In the experiment, an artificial operation that takes 500 μs to be processed on a specific server (Sun Blade X6270 with 2x Intel Xeon E5540, total 8 cores at 2.53 GHz, 24 GB RAM, ZFS RAID, SunOS 5.1, Java HotSpot x86 Server VM 1.6) has been monitored. The response time deviation is minimal as an extensive warm-up phase has been carried out to saturate the JVM behavior, particularly the just-in-time compilation. In the experiment, the monitored operation was continuously executed by 15 concurrent threads (on a server with 8 hyper-threading cores). The boxplots show that (1) instrumentation, i.e. processing previously woven, but inactive dummy probes, causes negligible overhead (Δ_I is less than 1 μs) compared to (2) data collection and (3) logging, i.e. creating and persisting the monitoring records to a Monitoring Log (Δ_C and Δ_L are each about 4 μs). In case (3) of the experiment where logging is enabled, the records were written asynchronously into the local file system by a dedicated writer thread. This avoids a direct delay of the response time perceived by the system users. The remaining logging overhead is effected by the thread concurrency. The evaluation results suggest the conclusion that injecting probes at a multitude of measuring points is not critical as long as data collection and logging can be (de)activated systematically. This finding underpins the adaptive activation of measuring points proposed in Section 3.4.

For this experiment setup, the total relative monitoring overhead is comparatively low with 1.5% ($= 7.7 \mu\text{s} / 500 \mu\text{s}$). In fact, this may not be announced as an universal result, as it decisively depends on the average response times of the monitored operations, the underlying execution environment, and the concurrency situation at runtime. A general recommendation is to configure a thread pool size that is smaller than the number of effectively available server cores, at least by one. The free core(s) can be used for asynchronous writing to a Monitoring Log or other parallel auxiliary tasks such as garbage collection.

5.3. Monitoring and Analysis Cost Evaluation

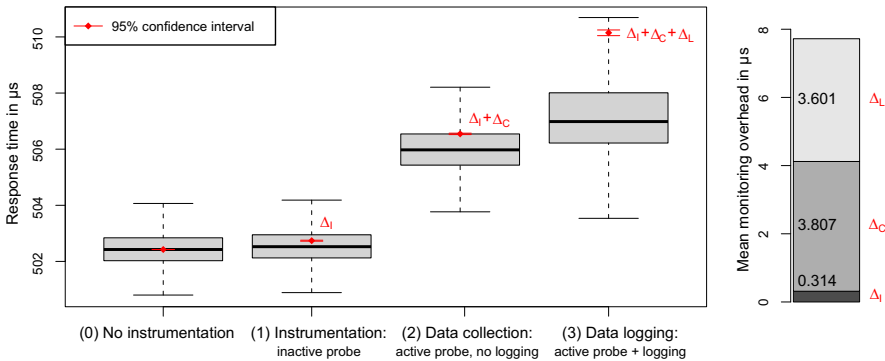


Figure 5.8. Evaluation of the monitoring cost [Ehlers et al., 2011]

5.3.2 Analysis Cost

The Kieker Analysis component is designed to be run on a dedicated server hosted in a local network connected to the servers of the monitored software system to ensure high data transfer rates. In the following experiment, a contemporary commercial off-the-shelf desktop computer (Acer Aspire X1700, Intel Core 2 Quad Q8200, 4 cores at 2.33 GHz, 4 GB RAM, Windows 7, Java HotSpot x86 Client VM 1.7) is employed to evaluate the Kieker Analysis component. The monitored software system is the SPECjEnterprise2010 installation described as Scenario F₂ in Section 5.1. The experiment is made up of 10 benchmark runs, each with a constant transaction rate during its 30 minutes runtime. The transaction rate is increased linearly from run to run (see Figure 5.9). As this exposes the monitored system to more and more workload, Kieker has to process a continuously increasing number of records/s. In this experiment, a full instrumentation of the SPECjEnterprise2010 is evaluated, i.e. probe measuring points are injected and activated for all operations in the namespace `org.spec.jent.*` (excluding getters, setters, and operations weaved due to enhancement of JPA entities). Applying Kieker's dynamic analysis capabilities (see Chapter 4), this leads to a CCT containing

5. Experimental Evaluation

73 different operations observed in 257 total calling contexts with averages of 5 operations/class and 35 operation executions/request (trace).

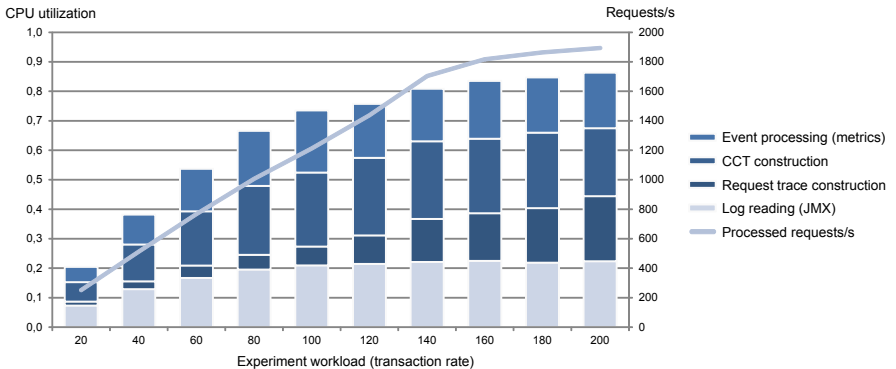


Figure 5.9. Evaluation of the analysis cost

In Figure 5.9, the computational analysis costs are quantified: The bar chart illustrates the affected CPU utilization on the analysis computer in relation to the overlaid line chart showing the increasing number of requests/s to be processed during the experiment runs. The analysis cost is decomposed into four parts: log reading (JMX), request trace construction, CCT construction, and event processing (metrics). Each part is designed to be executed by at least one separate JVM thread in order to utilize common multi-core CPUs. This allows for tracking the times how long the threads of each part have consumed CPU time, have been blocked by other threads for synchronization purposes, and waited on other threads to be notified and awakened (by means of the JMX interface `java.lang.management.ThreadMXBean`).

- Log reading (JMX): This part summarizes the computation effort of multiple threads related to data provision. A JMX-based monitoring log is used (JMXPersistence, see Section 4.1). As the JMX notification handler thread is a potential bottleneck, it should not block. Thus, record forwarding is delegated to temporarily blocking worker threads. By consequence, the following threads are involved: a log

5.3. Monitoring and Analysis Cost Evaluation

reader's main thread used to start and stop the filter from the GUI, a JMX notification handler thread that continuously receives incoming JMX notifications (by means of underlying RMI calls), and a pool of worker threads that pipe the records to the subscribed filters.

- Request trace construction: This part comprises a single-threaded TraceAnalysis filter, as described in Section 4.2.
- CCT construction: This part comprises the single-threaded construction of a CCT by the PerformanceAnomalyAnalysis plugin. Additionally, it contains sending of events to the Esper event processing library to update the associated performance metrics (see Section 4.3.6).
- Event processing (metrics): Esper is configured to use a pool of separate worker threads to process the sent events and recompute the metrics' values. In this experiment setup, the performance metrics ResponseTimeSummaryStatistics, Throughput, and TimeSeriesBasedAnomaly have been activated.

In the later experiment runs, the request throughput (processed requests/s) clearly saturates. The CPU capacity of the analysis computer is nearly fully utilized. The rest of the CPU capacity is spend for JVM background jobs such as garbage collection or remains unused due to thread synchronization.

The major finding of this experiment is Kieker's ability to process about 70,000 operation execution records/s ($35 \text{ records/request} \cdot 2000 \text{ requests/s}$) though a low priced desktop computer is used for analysis. If this is not sufficient, the proposed self-adaptive monitoring approach is an evident solution to adjust the number of active operation execution measuring points. This allows controlling the number of monitoring records to be processed in dependence of the workload to be handled.

5. Experimental Evaluation

5.4 Integrated Self-Adaptive Monitoring Evaluation

Compared to the monitoring and analysis cost, the additional computational cost caused by the continuous evaluation of the monitoring rules and the (de)activation of probe measuring points is marginal.

In this section, the feasibility of the self-adaptive monitoring approach is demonstrated in an integrated experiment setup using the anomaly scoring procedures having previously been evaluated in Section 5.2. The research question is: “Is a self-adaptive control of the monitoring coverage feasible in practice?” There is no single metric affirming this question. Instead, it is shown that the self-adaptive monitoring approach supports the detection of root causes for application-level software system anomalies.

Understandably, it is not possible to inject anomalies into a productive system of an industrial partner to test the approach. Thus, the evaluation is limited to lab scenarios where the cause-and-effect chains are known in advance, as the anomalies are caused on purpose. It is traced that the self-adaptive monitoring control in fact is able to zoom into a component and finally activate monitoring for those operations that are the root causes of the injected anomalies.

Experiment setup: The experiment comprises two scenarios, M_1 and M_2 . In both scenarios, the monitoring rules R_3 and R_4 as described in Section 4.4 are applied as part of the `MonitoringAdaptation` plugin to realize the self-adaptive monitoring control. The rule evaluation interval is set to 1 s and the activation freeze time to 15 s. The underlying anomaly scoring procedures are parametrized the same as described in Section 5.2. For each operation of both scenarios, a normal response time behavior has been learned in advance of the experiment run (by temporary activation of all measuring points or by employing the random activation monitoring rule R_5 described in Section 4.4).

5.4. Integrated Self-Adaptive Monitoring Evaluation

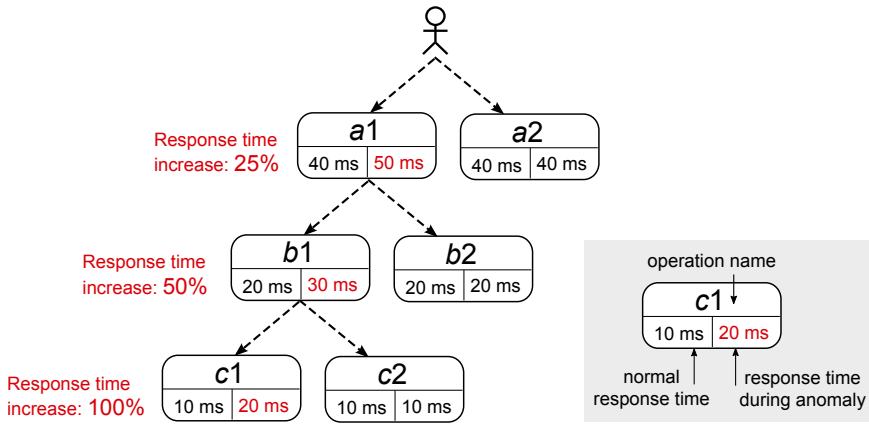


Figure 5.10. Call graph of sample system for anomaly injection

In Scenario M_1 , a synthetic sample system is studied, which provides 2 interface operations and 4 more internal operations. Figure 5.10 shows the system's call graph and the operations' response times in normal and anomalous operation. All response times vary with a 20% coefficient of variation. During anomalous operation, the response time of operation $c1$ is doubled causing a response time increase of 50% for operation $b1$ and 25% for $a1$ upwards in the call stack.

Scenario M_2 uses the JPetStore reference application (see Figure 5.1) again. An anomaly is injected by deleting a database index on the shop items' database table. By consequence, the service `viewProduct` increases in its response times as it depends on internal operations which query the affected database table. This anomaly typifies a common phenomenon in practice: A developer cleans up parts of an implementation and thereby removes a non-functional feature without overlooking its performance impacts, as it has not been documented or understood.

Experiment results: Figure 5.11 illustrates the reaction time of the self-adaptive monitoring approach referring to Scenario M_1 . For each

5. Experimental Evaluation

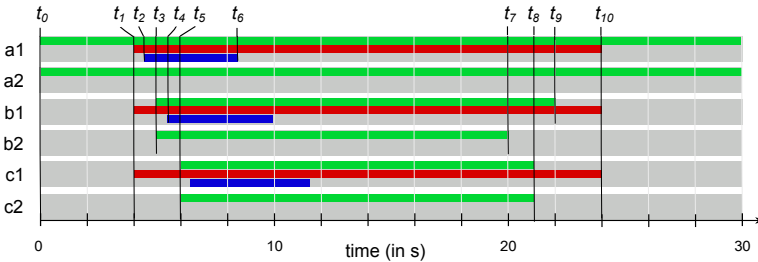
operation, a timeline depicts the intervals when the anomalous behavior was induced (red lines), when the operation's anomaly score exceeded the monitoring rule's threshold value (blue lines), and when the operation was effectively monitored on server-side (green lines). As stated above, the delay concerning the monitoring rule evaluation and the server-side measuring point activation is marginal and thus excluded for conciseness. How long it takes from the moment where the anomaly appears until the monitoring coverage is intensified depends on the used anomaly scoring procedure and its parametrization.

In the following, the evaluation results are discussed based on the chronological order of incidents marked in Figure 5.11 as t_0 - t_{11} :

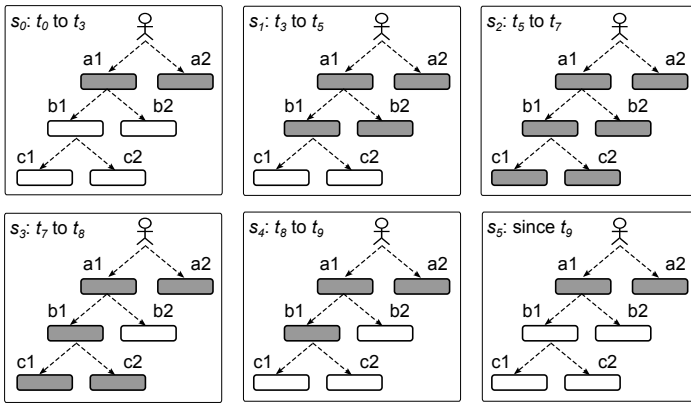
- t_0 : The experiment starts with the interface operations $a1$ and $a2$ being activated for monitoring. The experiment's runtime is 30 s only. The time spreads depicted in Figure 5.11 and commented below are average values of multiple experiment replications.
- t_1 : The operations $a1$, $b1$, and $c1$ begin to respond anomalous for 20 s until t_{10} .
- t_2 : The anomalous responsiveness of operation $a1$ is detected, insofar as its anomaly score exceeds the monitoring rule's threshold value. This takes 0.64 s with D-AS and 2.71 s with TS-AS in average (from t_1 to t_2). The difference in the reaction delay of about 2 s with TS-AS is mainly caused by a low, but reasonable anomaly score smoothing factor of 0.2, in comparison to a moving average window of 3 enabling an instantaneous reaction with D-AS (cf. Sections 3.4.3 and 4.3.6). In both cases, a delay of about 0.5 s is expected due to an interval of 1 s, in which operations executions occur equally distributed and are collected, averaged, and forwarded by the event processing engine. The marginal rest of the delay is owed to data processing and synchronization of the Kieker Analysis plugins.
- t_3 : The measuring points for operations $b1$ and $b2$ are activated, as these operations are known callees of the anomalous operation $a1$. The activation of the measuring points is the consequence of the monitoring rules' evaluation cycle.

5.4. Integrated Self-Adaptive Monitoring Evaluation

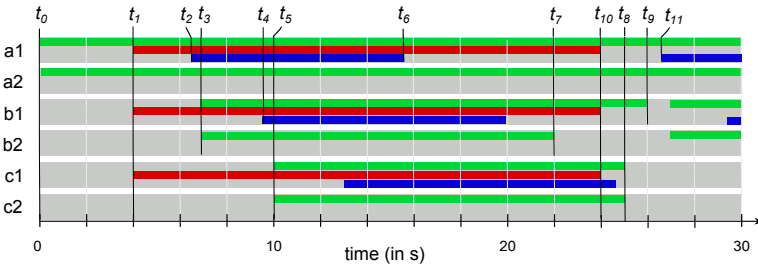
Distribution-based anomaly scoring



Monitoring coverage states



Time series-based anomaly scoring



- = operation actually monitored
- = anomalous operation behavior
- = anomaly score exceeds threshold
- = operation monitoring activated
- = operation monitoring deactivated
- = operation call

Figure 5.11. Evaluation of the self-adaptive monitoring process: Monitoring coverage transitions over time (comparing distribution- und time series-based anomaly scoring)

5. Experimental Evaluation

Therefore, the monitoring coverage changes from state s_0 to s_1 , as shown in Figure 5.11. Irrespective of the anomaly scoring procedure, this takes 0.59 s in average (from t_2 to t_3). As the rule evaluation interval is set to 1 s, again an average delay of 0.5 s is expected. The rest is owed to the evaluation of the OCL expressions and the remote calls for server-side measuring point activation.

- t_4 : The anomalous responsiveness of operation $b1$ is detected. Corresponding to t_2 , this takes about 2 s longer with TS-AS than with D-AS.
- t_5 : Again about 0.5 s later, the monitoring rules are evaluated causing an activation of the measuring points for the operations $c1$ and $c2$. This provokes a change from the monitoring coverage state s_1 to s_2 .
- t_6 : The anomaly score for operation $a1$ falls below the monitoring rule's threshold. The responsiveness is no longer rated as anomalous. The LoOP algorithm in D-AS needs 4 to 5 samples (4.67 s in average) to classify the higher response times no longer as outliers. The expected response time in TS-AS adapts more slowly (after 10.23 s in average) due to a small SES factor of 0.1.
- t_7 : Monitoring of operation $b2$ is deactivated after the activation freeze time of 15 s has elapsed. Although operation $b1$ is no longer rated anomalous as well, it cannot be deactivated as long as its callees are still monitored. Thus, the monitoring coverage changes from state s_2 to s_3 .
- t_8 : Monitoring is deactivated for the operations $c1$ and $c2$, since they are no longer rated anomalous and their activation freeze time has elapsed.
- t_9 : Finally, monitoring of operation $b1$ is deactivated a rule evaluation cycle later, i.e. 1 s after t_9 .
- t_{10} : The injected anomaly ends. The operations $a1$, $b1$, and $c1$ show normal responsiveness again. With D-AS, the monitoring coverage has already been completely reduced to the initial coverage, as the experiment's anomaly is very uniform and quickly understood as the new expected behavior. This motivates to setup a more reasonable activation freeze time in the range of some minutes in practice.

5.4. Integrated Self-Adaptive Monitoring Evaluation

- t_{11} : With TS-AS only, the end of the anomalous behavior causes another refinement of the monitoring coverage, as the expected response times have to be corrected back to the original normal behavior. With D-AS, this is not the case as its dataset capacity is large enough to restore the normal behavior instantly.

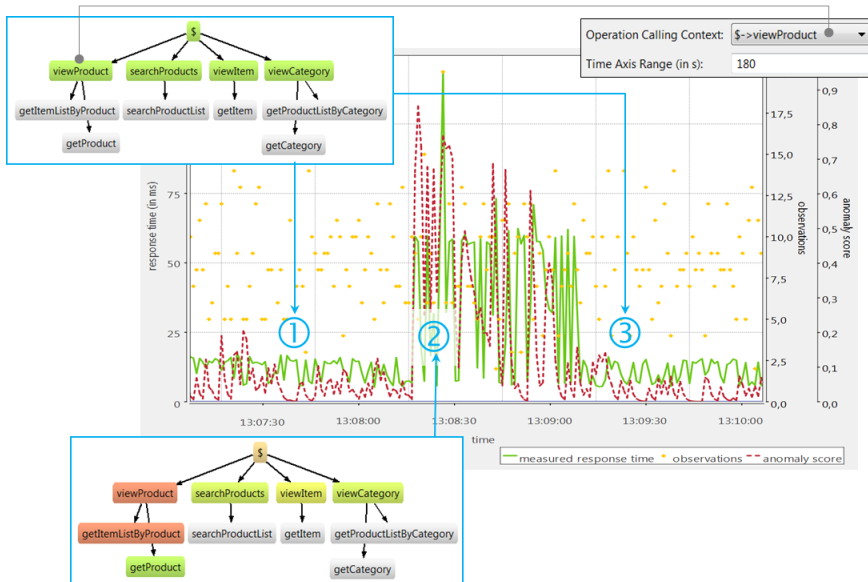


Figure 5.12. Self-adaptive monitoring process applied to the JPetStore application: Monitoring coverage before (1), during (2), and after (3) an anomaly incident

The aim of Scenario M_2 is to show that the self-adaptive monitoring approach does not work in a synthetic scenario only, but likewise in a typical enterprise software system, in which synchronous operation calls prevail. This is pointed out by the experiment results with the JPetStore application: Figure 5.12 contains a Kieker Analysis screenshot depicting a response time curve and an anomaly score curve for the `viewProduct` service that is affected by the injected anomaly. Furthermore, call graph screenshots are pinned onto the figure that illustrate the changing monitoring coverage

5. Experimental Evaluation

during experiment runtime. The call graph at the top shows the monitoring coverage before and after the anomaly has occurred. The call graph at the bottom shows the monitoring coverage during the anomaly, i.e. while the database index is removed for 1 min. As shown, the monitoring coverage is purposefully refined. The self-adaptive monitoring process allows the conclusion that in this case the operation `getItemListByProduct` is the anomaly's root cause (as it queries data from the `items`' table filtered by product with the corresponding index having been deleted). The responsiveness of the other services is slightly affected as a side effect due to more database utilization.

The experimental evaluation of the self-adaptive monitoring approach proved its practical feasibility in lab scenarios. The continuous evaluation of OCL-based monitoring rules affords a goal-oriented adaptation of the monitoring coverage at runtime. When an adoption to a productive system is planned, in particular the sensitivity of the anomaly scoring procedures has to be regarded. Certainly, there is no universal parametrization for these procedures that fits best for all systems. A thorough customization and test phase considering the individual characteristics of a specific software system and its administrators' and performance analysts' objectives is inevitable.

Related Work

Related work includes other approaches addressing application-level software system performance monitoring, particularly those containing self-adaptive features. Each integrated software system monitoring framework such as Kieker is concerned with two aspects: (1) monitoring, i.e. instrumentation and data acquisition, and (2) subsequent analysis. Section 6.1 discusses related research approaches, while Section 6.2 reviews the existing solutions at the commercial *application performance management (APM)* market.

6.1 Related Research Approaches

COMPAS JEEM: An integrated approach for tracing runtime paths in Java EE systems is provided by the COMPAS JEEM tool [Parsons et al., 2006, 2008; Diaconescu et al., 2004], which facilitates the instrumentation of probes at deployment time as a proxy layer to specific Java EE components. The instrumentation idea based on middleware interception is explained in detail in Section 2.3.3. In the context of COMPAS, adaptation of the monitoring coverage at runtime is studied in [Mos, 2004]. However, monitoring is restrained to the interface level of Java EE components such as EJBs or Servlets. By observing and analyzing the responsiveness of component-internal operations, Kieker allows a finer-grained insight.

6. Related Work

Magpie: The Magpie project¹ monitors request resource consumption and component interactions on component-level in distributed systems. Early versions of Magpie [Barham et al., 2003] used unique tokens to distinguish requests from each other. These tokens are propagated from one component to the next to reconstruct traces. In newer versions [Barham et al., 2004], the token passing has been replaced by a method based on events and timestamps in order to distinguish concurrent requests. In distributed systems, Magpie uses synchronization events between transmitted and received data to connect request data spanning multiple system nodes. For non-distributed systems, Kieker supports both timestamp-based and token-based trace distinction. In distributed systems, Kieker depends on token-passing while Magpie uses cross-machine synchronization events. Magpie has been implemented for Microsoft technology, while the Kieker implementation originally concentrates on Java technology.

Rainbow: The Rainbow project [Garlan et al., 2004] utilizes model-based monitoring in the context of architecture-based adaptation of software systems. Application-level data such as workload or response times is collected by probes. The system's architectural model is kept up-to-date at runtime applying a concept of so called *gauges*. A gauge aggregates the low-level data delivered by the probes and provides it to higher-level models. Appropriate self-adaptation steps concerning architectural component assembly are determined and executed once a model constraint violation is detected. In comparison to the presented self-adaptive monitoring approach with Kieker, monitoring and self-adaptation in Rainbow target capacity planning instead of QoS problem diagnosis. Similar to the SLastic approach [van Hoorn et al., 2009a] related to Kieker, Rainbow addresses dynamic capacity planning by automated architecture changes at runtime.

Pinpoint: The dynamic analysis approach Pinpoint [Chen et al., 2002] and its follow-up publications [Kiciman and Fox, 2005; Candea et al., 2006] perform runtime monitoring to support QoS problem diagnosis. Pinpoint tags client requests moving through the system and correlates anomalous patterns in the request traces to infer which components most

¹ <http://research.microsoft.com/en-us/projects/magpie/>

6.1. Related Research Approaches

likely cause the anomalies. In contrast to Kieker, Pinpoint does not collect performance time series, but applies pattern-oriented data mining techniques to detect anomalies in the requests. Pinpoint does not contribute means for self-adaptation to control the monitoring coverage.

Adaptive Monitoring by Munawar et al. [2008]: A further related approach addressing adaptive monitoring of component-based systems is presented in [Munawar et al., 2008; Munawar, 2009]. As shown in Figure 6.1, the approach distinguishes three different monitoring levels: minimal, detailed, and tracing. By default, minimal monitoring is activated for each component. Minimal monitoring comprises basic metrics that are accessed via the components' management interface (by means of JMX) and allow the detection of SLO violations. When a violation is detected, detailed monitoring is automatically activated to validate the violation using invariant-correlation models. If the invariants confirm a potential error indicated by the previous SLO violation, tracing (by means of ARM) is enabled for the suspected component instance and for one of its peers still deemed healthy. Similar to Kieker's self-adaptive monitoring solution, this approach supports early fault diagnosis by dynamically refining the monitoring coverage. In comparison, it provides only a small set of static monitoring levels.

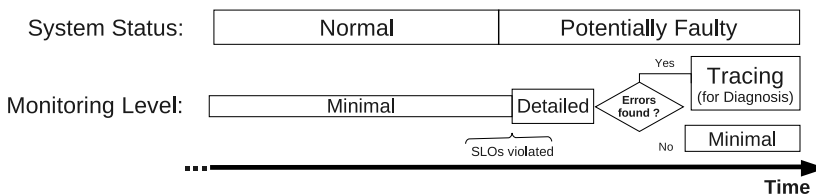


Figure 6.1. Monitoring levels of [Munawar et al., 2008]

Problem determination by Agarwal et al. [2004]: An early approach concerning performance anomaly localization in component-based systems is provided by [Agarwal et al., 2004]. The approach is based on predefined SLOs regarding response time thresholds for monitored user transactions.

6. Related Work

User transaction samples are classified to be either “good” or “bad” depending on whether the sample’s response time fulfills the corresponding SLO or not. Similarly, component-level service calls are subject to a binary classification, utilizing a dependency graph constructed by observing the transactions’ control flow. The classification rule is: If no parent transaction during a component call is in bad state, then the call is added to the component’s good behavior model and thereby impacts its dynamic SLO threshold. When a component-level SLO is violated, the corresponding component is pinpointed as a probable problem cause. This approach does not provide an integrated monitoring solution, but facilitates a method for performance anomaly localization and dynamic adaptation of the learned timing behavior.

Time Will Tell – Time spectra: Yilmaz et al. [2008] contribute another fundamental fault localization approach called Time Will Tell (TWT) that is based on method-level response time distributions. In the context of TWT, these are called time spectra. Time spectra are collected from passing and failing program executions. Behavior models are created using the time spectra of passing executions. Deviations from the learned behavior models in failing executions that take a suspicious amount of time are identified and scored as potential failure causes. Further related approaches are based on other abstractions of a program executions (called program spectra instead of time spectra) such as statements executed, branches covered, or call sequences [Jones et al., 2002; Renieres and Reiss, 2003].

Artemis: The Artemis approach [Fei and Midkiff, 2006] addresses selective and adaptive monitoring to reduce the effective monitoring overhead. It is argued that repeatedly executing a program region with the same context tends to produce the same outcome (classified as “correct” or “buggy”). The correct behavior is learned in sampling mode, with the sampling coverage being adjusted to system load. Artemis’ key strategy is to avoid remonitoring the same program region in the same context, as long as no buggy behavior is detected. The evaluation is done with C programs and libraries, whereat the context is limited to the state of simple type in-scope variables and method arguments. In consideration of pointers (or

6.2. Application Performance Management

object references) impacting a region's state, this is an imprecise context approximation.

Software monitoring with controllable overhead (SMCO): The SCMO approach [Huang et al., 2011] introduces the idea to restrain the monitoring overhead to a user-specified threshold. Monitoring of selected events can temporarily be disabled at runtime, as short as possible while still complying with a fixed max. target overhead. By this means, as many events are monitored as possible within in the given bounds. The overhead is controlled by employing strategies for a nonlinear control problem modeled as a composition of timed automata. SCMO differs from Kieker and other monitoring approaches, as its overhead does not vary in dependence of the monitored system's load and behavior. The evaluation of SCMO is so far limited to small, natively compiled programs which are not managed by a virtual runtime.

Software-EKG: The Software-EKG [Weigend et al., 2011] is an approach to integrate application- and infrastructure monitoring data. Data from infrastructure monitors (such as Nagios and the Windows Performance Counter), from middleware monitoring interfaces (such as JMX), and application-level log data (e.g. using Kieker) is extracted to flat files and imported into an analytics database. The goal is to correlate and visualize time-based events from the different log files and derive new insights from that. The Software-EKG assesses Kieker to be a promising tool for collecting application-level tracing data. While Kieker can be employed for continuous online analysis, the Software-EKG targets later offline analyses.

6.2 Application Performance Management

Application Performance Management (APM) is commonly used as a marketing term to categorize and promote services and related tools that focus on monitoring and controlling the performance and availability of software services. An appropriate definition is given by Menascé [2002].

6. Related Work

Definition: Application Performance Management by Menascé [2002]

APM is a collection of management processes used by organizations to ensure that the QoS of their e-business applications meets the business goals.

Accordingly, APM is associated with a lifecycle view of a software system (i.e. one or more applications and the employed resources) and supporting management processes to monitor, diagnose, report, and enhance the performance of that system. Sydor [2010] points out that the upcoming term *application performance monitoring*, which vendors often use synonymously with the same acronym, has a much more limited scope on software solutions, particularly tools, contributing to effective application-level software system monitoring. A list of such commercial tools is collected in Table 6.1.

Product	Vendor	Relevant Acquisitions
BMC Application Performance Management	BMC Software	Coradiant 2011, Identify 2006
CA Wily APM (former Wily Introscope)	CA Technologies	Wily 2006
DynaTrace	Compuware	DynaTrace 2005
HP (former Mercury) Diagnostics	HP	Mercury 2006
IBM Tivoli Composite Application Manager	IBM	Cyanea 2004, Candle 2004
OPNET APMXpert (former Altaworks Panorama)	OPNET Technologies	Altaworks 2004
Oracle Enterprise Manager	Oracle	ClearApp 2008
Quest Foglight	Quest Software	Foglight 2000

Product	Vendor	Founded
AppDynamics	AppDynamics	2008
New Relic	New Relic	2008
Prelerit	Prelerit	2008
JXInsight/OpenCore	JINSPiRED BV	2008
CorrelSense SharePath	CorrelSense	2005
OpTier Application Diagnostics	OpTier	2002
Nastel Autopilot M6	Nastel Technologies	1994
Precise	Precise Software Solutions	1990

Table 6.1. Commercial application performance monitoring tools

Gartner market research states a sum of about 2 billion US\$ that has been spent worldwide on APM licenses and first-year maintenance contracts in

6.2. Application Performance Management

2011, being a 15% increase compared to the previous year [Cappelli and Kowall, 2011]. The APM market is shared between established software enterprises such as Compuware, CA, HP, IBM, Oracle, and BMC. These market leaders are challenged by dynamic and innovative start-ups such as AppDynamics, New Relic, or CorrelSense. In the past, similar challengers such as DynaTrace, Wily, and ClearApp have been acquired by the larger enterprises. Due to ongoing acquisitions, the market situation changes quickly. Gartner regularly analyzes the APM market. Figure 6.2 depicts their overall evaluation results for 2011. Details can be found in [Cappelli and Kowall, 2011]. As known, a survey comparing proprietary tools is difficult to design and easily exposed to biases of vendors, resellers, or other stakeholders.

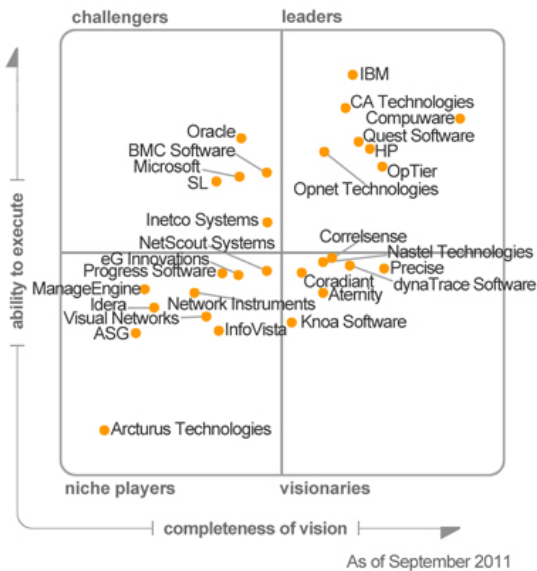


Figure 6.2. APM market as perceived by Gartner [Cappelli and Kowall, 2011]

Menascé [2002] distinguishes APM explicitly from performance benchmarking with competitors and performance testing of load capacities.

6. Related Work

Furthermore, he structures APM requirements into business-level, customer-level, and resource-level needs. A more contemporary categorization of APM requirements is provided by Gartner again:

1. End-user experience monitoring
2. Application runtime architecture discovery, modeling and display
3. User-defined transaction profiling
4. Component deep-dive monitoring in application context
5. Application performance analytics

Kieker addresses all of these requirement dimensions, although it is currently not able to compete with the commercial products regarding usability and professional marketing. Instead, Kieker is a research tool that targets the evaluation of new and prototype approaches for APM, particularly in lab experiments. This thesis concentrates on a self-adaptive monitoring approach based on operation-level anomaly scores, which can be sorted into the 4. and 5. dimension from above, i.e. application-level “deep-dive” monitoring and analytics. Self-learning capabilities including trend forecasting and complex event processing, as employed in this work, are notably identified by the market researchers as future APM needs. In the field, self-adaptation is currently based on less systematic and sophisticated monitoring policies than proposed in Chapter 3. Typically, a “deep dive” that activates call stack diagnoses is started when static, relative, or standard deviation-based thresholds for response times of specific request types are exceeded.

Additionally, recent APM advancements that are not in the scope of this thesis and remain future work for Kieker include the integration with cloud computing environments and the accompanied challenges of dynamic component assembly and redeployment at system runtime. SaaS (Software-as-a-Service) APM will relieve clients from operating a performance analysis control panel, while APM integrated into PaaS (Platform-as-a-Service) and IaaS (Infrastructure-as-a-Service) clouds will relieve clients from setting up the monitoring agents.

Conclusions and Future Work

This chapter provides the conclusions of the thesis. Section 7.1 summarizes the approach and its evaluation, which has been presented in detail in the previous sections. Section 7.1 discusses benefits and limitations of the approach and thereby infers the lessons learned. Remaining future work is suggested in Section 7.3.

7.1 Summary

This thesis contributes an approach for self-adaptive performance monitoring of component-based software systems. The approached solution includes a control feedback cycle, being typical for any autonomic system, that adapts the monitoring coverage at runtime. That is, probes and measuring points are automatically activated or deactivated to increase or decrease insight into the runtime behavior of system components. Guiding monitoring rules control the monitoring coverage, whereby the monitoring granularity is refined for those components that show anomalous responsiveness.

An overwhelming flood of monitoring data to be processed, filtered, and aggregated and the related computational overhead are the major arguments against full instrumentation of a software system during continuous operation. In contrast to profiling in test environments, productive environments require a founded decision where to instrument and

7. Conclusions and Future Work

activate which monitoring probes. The presented self-adaptive monitoring approach (Section 3.2) automatically collects more fine-grained data for those components and services that tend to be affected by performance problems. The monitored data includes performance measures such as service throughput and response time statistics, the utilization of underlying resources, as well as the inter- and intra-component control flow.

The monitoring rules are based on the OCL and refer to performance metrics that change their values at runtime (Section 3.3). Derived from past responsiveness samples, performance anomaly scores for each service and component-inherent operation are calculated. Two different anomaly scoring approaches are developed – based on time series forecasts and distribution clustering, respectively (Section 3.4). These anomaly scores allow a valuation whether a software service has recently behaved anomalous or not.

In practice, the integration of systematic monitoring capabilities at design time is still undervalued (Section 1.1). Reactive instrumentation of probes at static measuring points and manual debugging activities are time-consuming and stall the diagnosis of system failures. This business-critical failure diagnosis time can be reduced with the help of the proposed self-adaptive monitoring approach.

Tooling for the autonomic monitoring is provided by the Kieker monitoring framework, which has been advanced in the course of this research work (Chapter 4). The self-adaptive monitoring approach and the anomaly scores have been extensively evaluated in lab experiments (Chapter 5). The evaluation results demonstrate the feasibility and the practicability of the approach.

7.2 Lessons Learned

The lessons learned are subdivided in the following five aspects: (1) runtime architecture reconstruction, (2) monitoring rules and adaptation, (3) runtime adaptation of the monitoring coverage, (4) performance anomaly scores, and (5) tracing of service calls.

Runtime architecture reconstruction: AOP and BCI libraries (such as AspectJ and Javassist) provide efficient means for application performance monitoring tools (such as Kieker) to instrument modern component-based software systems. With the help of the instrumented monitoring probes, Kieker is able to reconstruct the effective assembly and interconnections of the system components at runtime – as far as possible in regards to reflection and tracing capabilities. Concerning Java- and .NET-based systems, reflection allows a simple determination of used packages, classes, operations and deployment artifacts. However, packages or deployment artifacts do not have to correspond to the components in the design models, e.g. match a UML component diagram. This mismatch between common design models and the available structures of the target programming languages causes a limitation for dynamic architecture reconstruction regarding component detection. A solution is model-driven annotation of component parts and interfaces, as proposed in Section 4.1.

According to expectation, runtime monitoring detects only those operations, classes, etc. that are affected by the processed user requests. When dynamically reconstructed architecture models are compared with design-time models or models won by static reverse engineering, the missing, i.e. unused, parts provide hints for unreachable code from the users' perspective. As the effective user workload might not cover all relevant parametrization combinations of the provided system services, it is advisable to schedule and continuously execute complete test suites on the productive system [Metzger et al., 2010].

7. Conclusions and Future Work

Tracing of service calls: Tracing of remote service calls is limited to technologies that allow client-side and server-side interception. Regarding web service calls for instance, request and response calls are intercepted to add tracing meta-data to the SOAP header (in case of JAX-WS) or to the HTTP header (in case of JAX-RS). For now, Kieker supports failure diagnosis for synchronous operation calls. Asynchronous calls, which respond immediately and process the actual task in a concurrent thread, are recently not correlated with their caller. This limitation can easily be resolved for asynchronous remote calls (e.g. via JMS) or local executor services that allow interception to attach meta-data to the call and its response.

Monitoring rules: The conditions of the monitoring rules are specified via OCL expressions. OCL is a well-known standardized language. Its built-in collection operations provide powerful ways of projecting or querying new data collections from existing ones. It is possible to evaluate arbitrary valid OCL expressions on EMF model instances. Tooling for the monitoring rule editor including code assist and syntax highlighting could easily be implemented, as described in Section 4.4. A drawback is the complexity of OCL expressions. The creation of a DSL, e.g. using Xtext, is an alternative that remained unimplemented. The advantage of a DSL is that it removes the complexity of OCL and is tailored to the specific requirements of the monitoring rules. However, it would produce another isolated language having to be explicitly learned without any chance of reuse.

Runtime monitoring coverage adaptation: The runtime adaptation of the monitoring coverage is resolved convincingly. Different technologies are appropriate to provide the remote adaptation services of a monitoring agent, e.g. in case of Java plain RMI, JMX, or JAX-RS (REST-based web services). Kieker's `MonitoringAdaptation` plugin makes use of JMX/RMI. The additional overhead is negligible (see Section 5.3).

Performance anomaly scores: The time series- and distribution-based performance anomaly scoring algorithms provide satisfying anomaly detection rates (see Section 5.2). However, both algorithms are subject to

sensitive parametrization to adjust them for an individual monitored system and the goals of the corresponding performance analyst. A limitation is that both algorithms depend in their quality on the scheduling discipline of the system's resource bottlenecks, as illustrated in Figure 5.4. That is, the proposed and studied anomaly detection algorithms behave worse in highly multi-threaded and interdependent systems. Decisive context-determinant impact factors on a service's response time are the number and the types of concurrent requests or transactions. A separation of the concurrent workload intensity in a further contextual dimension (in addition to the call stack) has not been evaluated in this thesis. A basic approach is presented in the related work of Rohr et al. [2010]. Furthermore, there exist more anomaly detection techniques [Han et al., 2011; Theodoridis and Koutroumbas, 2008] that can be adopted to score the timing behavior of the software services in future work.

7.3 Future Work

Partially, future work has already been discussed in the lessons learned above. For instance, the idea of model-driven annotation of architectural composition elements, in particular component interfaces, has been named. This idea can be integrated into a principal model-driven development approach, including generation of instrumentation and further meta-data related to performance (e.g. markup of performance-critical parameters). An outlook is given by van Hoorn et al. [2011b].

As well, the opportunity to develop a DSL for the specification of the monitoring rules has been suggested above. A more concise DSL that restrains the complexity of OCL can improve the usability of the Kieker MonitoringAdaptation plugin.

Furthermore, a more fine-grained separation of context-determinant performance impact factors remains future work concerning the Kieker framework. This includes monitoring and analyzing different levels

7. Conclusions and Future Work

of workload intensity and concurrent transactions to consider these for performance anomaly scoring.

In the scope of this thesis, only Java-based software systems have been monitored and evaluated. Current and future work targets monitoring other runtime platforms and programming languages such as the .NET CLR and its predecessors [van Hoorn et al., 2011a].

The integration of novel visualization approaches, addressing the comprehension of software system behavior for example as “interactive sequence diagrams” [Grati et al., 2010], “3D code cities” [Wettel et al., 2011], or “circular bundle views” [Cornelissen et al., 2011], is current or future work. The creators of these new visualizations attest the added value of their approaches by controlled experiments. If their results are convincing, the integration into active monitoring frameworks such as Kieker can support the spreading and acceptance of innovative research.

An important aspect of future work will tackle the evaluation of the self-adaptive monitoring approach in productive systems of industry partners. For that, trust in the basic monitoring capabilities of Kieker and our research group is a necessary prerequisite. The monitoring tool has to be run continuously for a longer period of time to observe real failures or performance problems during operation. Certainly, manual fault injection is no option for productive systems. Regarding the publishing activities as a desired outcome of research work, the problem of negative reputation effects falling back on the industry partners, in case serious failures or performance problems are detected and analyzed, has to be considered and clarified.

Bibliography

- [Achtert et al., 2010] Achtert, E., Kriegel, H.-P., Reichert, L., Schubert, E., Wojdanowski, R., and Zimek, A. Visual evaluation of outlier detection models. In *15th International Conference on Database Systems for Advanced Applications*, volume 5982 of *Lecture Notes in Computer Science*, pages 396–399, Tsukuba, Japan. Springer, 2010.
- [Agarwal et al., 2004] Agarwal, M. K., Appleby, K., Gupta, M., Kar, G., Neogi, A., and Sailer, A. Problem determination using dependency graphs and run-time behavior models. In *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, volume 3278 of *Lecture Notes in Computer Science*, pages 171–182. Springer, 2004.
- [Agrawal et al., 1997] Agrawal, R., Gupta, A., and Sarawagi, S. Modeling multidimensional databases. In *Proceedings of the 13th International Conference on Data Engineering (ICDE '97)*, pages 232–243, Birmingham, UK. IEEE Computer Society, 1997.
- [Akai and Chiba, 2009] Akai, S. and Chiba, S. Extending AspectJ for separating regions. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE '09)*, pages 45–54. ACM, 2009.
- [Allen and Garlan, 1997] Allen, R. and Garlan, D. A formal basis for architectural connection. *ACM Transactions on Software Engineering Methodologies*, 6(3):213–249, 1997.
- [Alpern et al., 2005] Alpern, B., Augart, S., Blackburn, S. M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K. S., Mergen, M., Moss, J. E. B., Ngo, T., and Sarkar, V. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

Bibliography

- [Ambler et al., 2005] Ambler, S. W., Vizdos, M. J., and Nalbhone, J. *The Enterprise Unified Process: Extending the Rational Unified Process*. Prentice Hall, 2005.
- [Ammons et al., 1997] Ammons, G., Ball, T., and Larus, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, USA. ACM Press, 1997.
- [Arisholm et al., 2004] Arisholm, E., Briand, L. C., and Foyen, A. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30:491–506, 2004.
- [Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
- [Balsamo et al., 2004] Balsamo, S., Di Marco, A., Inverardi, P., and Simeoni, M. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
- [Banks et al., 2009] Banks, J., Carson, J. S., Nelson, B. L., and Nicol, D. M. *Discrete Event System Simulation*. Pearson Education, 5th edition, 2009.
- [Baresi et al., 2008] Baresi, L., Guinea, S., and Tamburrelli, G. Towards decentralized self-adaptive component-based systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '08)*, pages 57–64, Leipzig, Germany. ACM, 2008.
- [Barham et al., 2004] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems*

Design & Implementation (OSDI '04), pages 259–272, Berkeley, CA, USA. USENIX, 2004.

- [Barham et al., 2003] Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems – Vol. 9*, page 15, Lihue, HI, USA. USENIX, 2003.
- [Basili, 1992] Basili, V. Software modeling and measurement: The goal/question/metric paradigm. Technical Report CS-TR-2956, University of Maryland, 1992.
- [Bause and Kritzinger, 2002] Bause, F. and Kritzinger, P. S. *Stochastic Petri Nets*. Vieweg + Teubner, 2nd edition, 2002.
- [Becker, 2008] Becker, S. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. KIT Scientific Publishing, 2008.
- [Becker et al., 2006] Becker, S., Grunske, L., Mirandola, R., and Overhage, S. Performance prediction of component-based systems - a survey from an engineering perspective. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 169–192, Dagstuhl, Germany. Springer, 2006.
- [Becker et al., 2007] Becker, S., Kozirolek, H., and Reussner, R. Model-based performance prediction with the Palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance*, pages 54–65, Buenos Aires, Argentina. ACM Press, 2007.
- [Becker et al., 2009] Becker, S., Kozirolek, H., and Reussner, R. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.
- [Bernardo and Hilston, 2007] Bernardo, M. and Hilston, J., editors. *Formal Methods for Performance Evaluation (7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems)*, volume 4486 of *Lecture Notes in Computer Science*. Springer, 2007.

Bibliography

- [Bertoli et al., 2009] Bertoli, M., Casale, G., and Serazzi, G. JMT: Performance engineering tools for system modeling. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.
- [Birkmeier and Overhage, 2009] Birkmeier, D. and Overhage, S. On component identification approaches — classification, state of the art, and comparison. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE '09)*, pages 1–18, East Stroudsburg, PA, USA. Springer, 2009.
- [Bishop, 2006] Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2nd edition, 2006.
- [Boehm, 1984] Boehm, B. W. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):77–88, 1984.
- [Boehm, 1988] Boehm, B. W. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [Bolch et al., 2006] Bolch, G., Greiner, S., de Meer, H., and Trivedi, K. S. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation With Computer Science Applications*. Wiley, 2nd edition, 2006.
- [Booch, 1994] Booch, G. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2nd edition, 1994.
- [Box et al., 2008] Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. *Time Series Analysis: Forecasting and Control*. Wiley, 4th edition, 2008.
- [Breunig et al., 2000] Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. LOF: Identifying density-based local outliers. *ACM SIGMOD Record*, 29:93–104, 2000.
- [Briand et al., 1999] Briand, L. C., Daly, J. W., and Wüst, J. K. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25:91–121, 1999.

- [Bruneliere et al., 2010] Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*, pages 173–174. ACM, 2010.
- [Bruneton et al., 2002] Bruneton, E., Lenglet, R., and Coupaye, T. ASM: a code manipulation tool to implement adaptable. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002: Adaptable and extensible component systems*, Grenoble, France, 2002.
- [Buzen, 1971] Buzen, J. P. *Queueing Network Models of Multiprogramming*. PhD thesis, Harvard University, Massachusetts, 1971.
- [Candea et al., 2006] Candea, G., Kiciman, E., Kawamoto, S., and Fox, A. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9:175–190, 2006.
- [Cappelli and Kowall, 2011] Cappelli, W. and Kowall, J. Magic quadrant for application performance monitoring. <http://www.gartner.com/technology/streamReprints.do?id=1-17DC04V&ct=110920>, 2011.
- [Chakravarti and Baumgartner, 2004] Chakravarti, A. J. and Baumgartner, G. The organic grid: Self-organizing computation on a peer-to-peer network. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC '04)*, pages 96–103, New York, NY, USA. IEEE Computer Society, 2004.
- [Chandola et al., 2009] Chandola, V., Banerjee, A., and Kumar, V. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58, 2009.
- [Chaudhuri and Dayal, 1997] Chaudhuri, S. and Dayal, U. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.
- [Cheesman and Daniels, 2000] Cheesman, J. and Daniels, J. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison Wesley, 2000.

Bibliography

- [Chen et al., 2002] Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Bethesda, MD, USA. IEEE Computer Society, 2002.
- [Cheng et al., 2008] Cheng, B. H., Giese, H., Inverardi, P., Magee, J., de Lemos, R., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. Software engineering for self-adaptive systems: A research road map. In *Software Engineering for Self-Adaptive Systems*, number 08031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2008.
- [Chiba, 2000] Chiba, S. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*, pages 313–336, London, UK. Springer, 2000.
- [Chiba and Ishikawa, 2005] Chiba, S. and Ishikawa, R. Aspect-oriented programming beyond dependency injection. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP '05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143, Glasgow, UK. Springer, 2005.
- [Clements et al., 2002] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.
- [Cornelissen et al., 2011] Cornelissen, B., Zaidman, A., and van Deursen, A. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37:341–355, 2011.

- [Cortellessa et al., 2011] Cortellessa, V., Di Marco, A., and Inverardi, P. *Model-Based Software Performance Analysis*. Springer, 2011.
- [Cuesta et al., 2001] Cuesta, C. E., de la Fuente, P., and Barrio-Solárzano, M. Dynamic coordination architecture through the use of reflection. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC '01)*, pages 134–140, Las Vegas, NV, USA. ACM, 2001.
- [Dahm, 2001] Dahm, M. Byte code engineering with the bcel api. Technical Report B-17-98, Freie Universität Berlin, 2001.
- [Dashofy et al., 2005] Dashofy, E. M., van der Hoek, A. v. d., and Taylor, R. N. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering Methodologies*, 14(2):199–245, 2005.
- [Di Marzo Serugendo et al., 2007] Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A., and Guelfi, N. A generic framework for the engineering of self-adaptive and self-organising systems. Technical Report CS-TR-1018, University of Newcastle, 2007.
- [Diaconescu et al., 2004] Diaconescu, A., Mos, A., and Murphy, J. Automatic performance management in component based software systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC '04)*, pages 214–221, New York, NY, USA. IEEE Computer Society, 2004.
- [Dobson et al., 2006] Dobson, S., Denazis, S., Fernandez, A., Gaiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., and Zambonelli, F. A survey of autonomic communications. *ACM Transactions Autonomous Adaptive Systems (TAAS)*, 1(2):223–259, 2006.
- [Duda et al., 2000] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern Classification*. Wiley, 2nd edition, 2000.
- [Efftinge and Voelter, 2006] Efftinge, S. and Voelter, M. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit 2006*, 2006.

Bibliography

- [Ehlers and Hasselbring, 2011a] Ehlers, J. and Hasselbring, W. A self-adaptive monitoring framework for component-based software systems. In *5th European Conference on Software Architecture (ECSA '11)*, pages 278–286. Springer, 2011.
- [Ehlers and Hasselbring, 2011b] Ehlers, J. and Hasselbring, W. Self-adaptive software performance monitoring. In *Software Engineering 2011*, volume 183 of *Lecture Notes in Informatics*, pages 51–62. GI, 2011.
- [Ehlers et al., 2011] Ehlers, J., van Hoorn, A., Waller, J., and Hasselbring, W. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC '11)*, pages 197–200, Karlsruhe, Germany. ACM, 2011.
- [Fei and Midkiff, 2006] Fei, L. and Midkiff, S. P. Artemis: Practical runtime monitoring of applications for execution anomalies. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 84–95, Ottawa, Canada. ACM, 2006.
- [Feiler et al., 2003] Feiler, P. H., Lewis, B., and Vestal, S. The SAE avionics architecture description language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *Proceedings of Workshop on Model-Driven Embedded Systems*, Washington D.C., USA, 2003.
- [Fortier and Michel, 2003] Fortier, P. and Michel, H. *Computer systems performance evaluation and prediction*. Butterworth Heinemann, 2003.
- [Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [Garlan et al., 2004] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

- [Garlan et al., 2000] Garlan, D., Monroe, R. T., and Wile, D. Acme: architectural description of component-based systems. *Foundations of component-based systems*, pages 47–67, 2000.
- [Garlan and Schmerl, 2002] Garlan, D. and Schmerl, B. Model-based adaptation for self-healing systems. In *Proceedings of the 1st Workshop on Self-healing Systems (WOSS '02)*, pages 27–32, Charleston, SC, USA. ACM, 2002.
- [Gat, 1998] Gat, E. Three-layer architectures. *Artificial intelligence and mobile robots: case studies of successful robot systems*, pages 195–210, 1998.
- [Gelernter, 1985] Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985.
- [Georges et al., 2007] Georges, A., Buytaert, D., and Eeckhout, L. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.
- [Gorlick and Razouk, 1991] Gorlick, M. M. and Razouk, R. R. Using Weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE '91)*, pages 23–34, Austin, TX, USA. IEEE Computer Society, 1991.
- [Graham et al., 1982] Graham, S. L., Kessler, P. B., and Mckusick, M. K. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, 1982.
- [Grassi et al., 2007a] Grassi, V., Mirandola, R., and Sabetta, A. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528–558, 2007.
- [Grassi et al., 2007b] Grassi, V., Mirandola, R., and Sabetta, A. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In *Proceedings of the 6th International Workshop on Software and Performance (WOSP '07)*, pages 103–114, Buenos Aires, Argentina. ACM, 2007.

Bibliography

- [Grati et al., 2010] Grati, H., Sahraoui, H., and Poulin, P. Extracting sequence diagrams from execution traces using interactive visualization. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering (WCRE '10)*, pages 87–96, Washington, DC, USA. IEEE Computer Society, 2010.
- [Greenwood and Blair, 2006] Greenwood, P. and Blair, L. A framework for policy driven auto-adaptive systems using dynamic framed aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:30–65, 2006.
- [Gruschke, 1998] Gruschke, B. Integrated event management: Event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, Newark, DE, USA, 1998.
- [Haesevoets et al., 2010] Haesevoets, R., Truyen, E., Holvoet, T., and Joosen, W. Weaving the fabric of the control loop through aspects. In *1st International Workshop on Self-Organizing Architectures (SOAR '09)*, volume 6090 of *Lecture Notes in Computer Science*, pages 38–65, Cambridge, UK. Springer, 2010.
- [Han et al., 2011] Han, J., Kamber, M., and Pei, J. *Data Mining – Concepts and Techniques*. Morgan Kaufmann, 3rd edition, 2011.
- [Harbulot and Gurd, 2006] Harbulot, B. and Gurd, J. R. A join point for loops in AspectJ. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development (AOSD '06)*, pages 63–74. ACM, 2006.
- [Hauswirth et al., 2004] Hauswirth, M., Sweeney, P. F., Diwan, A., and Hind, M. Vertical profiling: understanding the behavior of object-oriented applications. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 251–269, Vancouver, Canada. ACM, 2004.

- [Hayes-Roth, 1985] Hayes-Roth, F. Rule-based systems. *Communications of the ACM*, 28(9), 1985.
- [Huang et al., 2011] Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S., Stoller, S., and Zadok, E. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–21, 2011.
- [IEEE Computer Society, 2000] IEEE Computer Society. IEEE Std 1471-2000 – IEEE recommended practice for architectural description of software-intensive systems. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=875998, 2000.
- [ISO, 2005] ISO. ISO 9000:2005 Quality management systems – Fundamentals and vocabulary. http://www.iso.org/iso/iso_catalogue/catalogue_detail.htm?csnumber=42180, 2005.
- [ISO/IEC, 2008] ISO/IEC. ISO/IEC 9075-(1-4,9-11,13,14):2008 Information technology – Database languages – SQL. http://www.iso.org/iso/iso_catalogue/catalogue_detail.htm?csnumber=45498, 2008.
- [ISO/IEC/IEEE, 2011] ISO/IEC/IEEE. ISO/IEC/IEEE 42010:2011 Systems and software engineering – Architecture description. http://www.iso.org/iso/catalogue_detail.htm?csnumber=50508, 2011.
- [Jackson, 2002] Jackson, D. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
- [Jacobson, 1992] Jacobson, I. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
- [Jacobson et al., 1999] Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, 1999.
- [Jain, 1991] Jain, R. *The Art of Computer Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

Bibliography

- [Jensen and Kristensen, 2009] Jensen, K. and Kristensen, L. M. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.
- [Jerding et al., 1997] Jerding, D. F., Stasko, J. T., and Ball, T. Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 360–370, Boston, MA, USA. ACM, 1997.
- [Johnson, 1998] Johnson, M. W. Monitoring and diagnosing application response time with ARM. In *Proceedings of the IEEE 3rd International Workshop on Systems Management (SMW '98)*, page 4. IEEE Computer Society, 1998.
- [Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 467–477, Orlando, FL; USA. ACM, 2002.
- [Jonkers et al., 2004] Jonkers, H., Lankhorst, M., van Buuren, R., Hoppenbrouwers, S., Bonsangue, M., and van der Torre, L. Concepts for modelling enterprise architectures. *International Journal of Cooperative Information Systems*, 13(3):257–287, 2004.
- [Keller and Ludwig, 2003] Keller, A. and Ludwig, H. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.
- [Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. The vision of autonomic computing. *Computer*, 26(1):41–50, 2003.
- [Kiciman and Fox, 2005] Kiciman, E. and Fox, A. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, 2005.
- [Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of aspectj.

In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, pages 327–353, London, UK. Springer, 2001.

- [Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
- [Kimball and Ross, 2002] Kimball, R. and Ross, M. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, 2nd edition, 2002.
- [Kleinrock, 1976] Kleinrock, L. *Queueing Systems*, volume 1, 2. Wiley, 1976.
- [Koschke, 1999] Koschke, R. An incremental semi-automatic method for component recovery. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pages 256–267, Atlanta, GA, USA. IEEE, 1999.
- [Kounev and Buchmann, 2003] Kounev, S. and Buchmann, A. Performance modelling of distributed e-business applications using queuing Petri nets. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '03)*, pages 143–155, Austin, TX, USA. IEEE, 2003.
- [Kounev and Dutz, 2009] Kounev, S. and Dutz, C. QPME: a performance modeling tool based on queueing petri nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):46–51, 2009.
- [Koziolok, 2008] Koziolok, H. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. KIT Scientific Publishing, 2008.
- [Koziolok, 2010] Koziolok, H. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

Bibliography

- [Koziolok et al., 2008] Koziolok, H., Becker, S., Happe, J., and Reussner, R. *Model-Driven Software Development: Integrating Quality*, chapter Evaluating Performance of Software Architecture Models with the Palladio Component Model, pages 95–118. IDEA Group Inc., 2008.
- [Kramer and Magee, 2007] Kramer, J. and Magee, J. Self-managed systems: an architectural challenge. In *Future of Software Engineering 2007 (FOSE '07)*, pages 259–268, Minneapolis, MN, USA. IEEE Computer Society, 2007.
- [Kriegel et al., 2009] Kriegel, H.-P., Kröger, P., Schubert, E., and Zimek, A. LoOP: Local outlier probabilities. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 1649–1652, Hongkong, China. ACM, 2009.
- [Kruchten, 1995] Kruchten, P. Architectural blueprints: The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, 1995.
- [Laddad and Johnson, 2009] Laddad, R. and Johnson, R. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications, 2nd edition, 2009.
- [Lau and Wang, 2005] Lau, K.-K. and Wang, Z. A taxonomy of software component models. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 88–95, Porto, Portugal. IEEE Computer Society, 2005.
- [Lee et al., 2001] Lee, J. K., Seung, S. J., Kim, S. D., Hyun, W., and Han, D. H. Component identification method with coupling and cohesion. In *Proceedings of the 8th Asia-Pacific on Software Engineering Conference (APSEC '01)*, pages 79–86, Macau, China. IEEE Computer Society, 2001.
- [Liehr and Buchenrieder, 2009] Liehr, A. W. and Buchenrieder, K. J. Simulating inter-process communication with extended queuing networks. *Simulation Modelling Practice and Theory*, 2009.
- [Lindholm and Yellin, 1999] Lindholm, T. and Yellin, F. *Java Virtual Machine Specification*. Prentice Hall, 2nd edition, 1999.

- [Luckham et al., 1995] Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., and Mann, W. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [Magee et al., 1995] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, London, UK. Springer, 1995.
- [Marwede et al., 2009] Marwede, N., Rohr, M., van Hoorn, A., and Hasselbring, W. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (CSMR '09)*, pages 47–57, Kaiserslautern, Germany. IEEE, 2009.
- [Marzolla, 2004] Marzolla, M. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD thesis, Universita Ca Foscari di Venezia, Italy, 2004.
- [Matevska, 2008] Matevska, J. An optimised runtime reconfiguration of component-based software systems. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC '08)*, pages 499–501, Turku, Finland. IEEE Computer Society, 2008.
- [May, 2005] May, N. A survey of software architecture viewpoint models. In *Sixth Australasian Workshop on Software and System Architectures*, pages 13–24, Brisbane, Australia. Swinburne University of Technology, 2005.
- [McKinley et al., 2004] McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. C. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

Bibliography

- [Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
- [Menascé, 2002] Menascé, D. A. Load testing, benchmarking, and application performance management for the web. In *Proceedings of the 2002 Computer Management Group Conference*, pages 271–281, Reno, NV, USA, 2002.
- [Menascé and Almeida, 2001] Menascé, D. A. and Almeida, V. A. F. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, 2001.
- [Metzger et al., 2010] Metzger, A., Sammodi, O., Pohl, K., and Rzepka, M. Towards pro-active adaptation with confidence: augmenting service monitoring with online testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*, pages 20–28, Cape Town, South Africa. ACM, 2010.
- [Meyer, 1997] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [Meyerhöfer, 2007] Meyerhöfer, M. TestEJB: response time measurement and call dependency tracing for EJBs. In *Proceedings of the 1st Workshop on Middleware-Application Interaction, in conjunction with Euro-Sys 2007*, pages 55–60, Lisbon, Portugal. ACM, 2007.
- [Montgomery and Runger, 2006] Montgomery, D. C. and Runger, G. C. *Applied Statistics and Probability for Engineers*. Wiley, 4th edition, 2006.
- [Mos, 2004] Mos, A. *A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications*. PhD thesis, Dublin City University, Ireland, 2004.

- [Mos and Murphy, 2004] Mos, A. and Murphy, J. COMPAS: Adaptive performance monitoring of component-based systems. In *2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04), 26th International Conference on Software Engineering (ICSE '04)*, pages 35–40, Edinburgh, UK, 2004.
- [Müller et al., 2008] Müller, H., Pezzè, M., and Shaw, M. Visibility of control in adaptive systems. In *Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS '08)*, pages 23–26, Leipzig, Germany. ACM, 2008.
- [Munawar, 2009] Munawar, M. A. *Adaptive Monitoring of Complex Software Systems using Management Metrics*. University of Waterloo, Canada, 2009.
- [Munawar et al., 2008] Munawar, M. A., Reidemeister, T., Jiang, M., George, A., and Ward, P. A. Adaptive monitoring with dynamic differential tracing-based diagnosis. In *Proceedings of the 19th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '08)*, pages 162–175, Samos, Greece. Springer, 2008.
- [Mytkowicz et al., 2010] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 10)*, Toronto, Canada. ACM, 2010.
- [Mytkowicz et al., 2008] Mytkowicz, T., Sweeney, P. F., Hauswirth, M., and Diwan, A. Observer effect and measurement bias in performance analysis. Technical Report CU-CS 1042-08, University of Colorado at Boulder, 2008.
- [Nielsen, 1993] Nielsen, J. *Usability Engineering*. Morgan Kaufmann, 1993.
- [Nisbet et al., 2009] Nisbet, R., Elder, J., and Miner, G. *Handbook of Statistical Analysis and Data Mining Applications*. Academic Press, 2009.

Bibliography

- [Oesterreich and Bremer, 2009] Oesterreich, B. and Bremer, S. *Analyse und Design mit UML 2.3 – Objektorientierte Softwareentwicklung*. Oldenbourg, 2009.
- [Okanovic et al., 2011] Okanovic, D., van Hoorn, A., Konjovic, Z., and Vidakovic, M. Towards adaptive monitoring of Java EE applications. In *Proceedings of the 5th International Conference on Information Technology (ICIT '11)*, Amman, Jordan, 2011.
- [OMG, 2005] OMG. UML Profile for Schedulability, Performance, and Time, v1.1. <http://www.omg.org/spec/SPTP/1.1/>, 2005.
- [OMG, 2008] OMG. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification, Version 1.1. <http://www.omg.org/spec/QFTP/1.1/>, 2008.
- [OMG, 2010] OMG. Object Constraint Language, Version 2.2. <http://www.omg.org/spec/OCL/2.2/>, 2010.
- [OMG, 2011a] OMG. Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), Version 1.3. <http://www.omg.org/spec/KDM/1.3/>, 2011.
- [OMG, 2011b] OMG. Business Process Model and Notation (BPMN), Version 2.0. <http://www.omg.org/spec/BPMN/2.0/>, 2011.
- [OMG, 2011c] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1. <http://www.omg.org/spec/QVT/1.1/>, 2011.
- [OMG, 2011d] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. <http://www.omg.org/spec/MOF/2.4.1/>, 2011.
- [OMG, 2011e] OMG. OMG Unified Modeling Language, Superstructure Version 2.4.1. <http://www.omg.org/spec/UML/2.4.1/>, 2011.
- [OMG, 2011f] OMG. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1. <http://www.omg.org/spec/MARTE/1.1/>, 2011.

- [Oreizy et al., 2008] Oreizy, P., Medvidovic, N., and Taylor, R. N. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion 2008: Companion of the 30th International Conference on Software Engineering*, pages 899–910, Leipzig, Germany. ACM, 2008.
- [Page and Kreutzer, 2005] Page, B. and Kreutzer, W. *The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java*. Shaker, 2005.
- [Parsons et al., 2006] Parsons, T., Mos, A., and Murphy, J. Non-intrusive end-to-end runtime path tracing for J2EE systems. *IEEE Proceedings – Software*, 153(4):149–161, 2006.
- [Parsons et al., 2008] Parsons, T., Mos, A., Trofin, M., Gschwind, T., and Murphy, J. Extracting interactions in component-based systems. *IEEE Transactions on Software Engineering*, 34(6):783–799, 2008.
- [Pacha and Park, 2007] Pacha, A. and Park, J.-M. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.
- [Petriu and Woodside, 2007] Petriu, D. B. and Woodside, M. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and Systems Modeling*, 6(2):163–184, 2007.
- [Pokrajac et al., 2007] Pokrajac, D., Lazarevic, A., and Latecki, L. J. Incremental local outlier detection for data streams. In *IEEE Symposium on Computational Intelligence and Data Mining*, pages 504–515. IEEE, 2007.
- [Renieres and Reiss, 2003] Renieres, M. and Reiss, S. P. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering 2003 (ASE '03)*, pages 30–39, Montreal, Canada. IEEE, 2003.
- [Rohr et al., 2006] Rohr, M., Giesecke, S., Hasselbring, W., Hiel, M., van den Heuvel, W.-J., and Weigand, H. A classification

Bibliography

- scheme for self-adaptation research. In *International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS06)*, Erfurt, Germany, 2006.
- [Rohr et al., 2008] Rohr, M., van Hoorn, A., Giesecke, S., Matevska, J., Hasselbring, W., and Alekseev, S. Trace-context sensitive performance profiling for enterprise software applications. In *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW '08)*, volume 5119 of *Lecture Notes in Computer Science*, pages 283–302, Darmstadt, Germany. Springer, 2008.
- [Rohr et al., 2010] Rohr, M., van Hoorn, A., Hasselbring, W., Lübcke, M., and Alekseev, S. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 87–92. ACM, 2010.
- [Ross, 2009] Ross, S. M. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press, 4th edition, 2009.
- [Royce, 1987] Royce, W. W. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering (ICSE '87)*, pages 328–338, Los Alamitos, CA, USA. IEEE Computer Society Press, 1987.
- [Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenzen, W. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [Sahai et al., 2002] Sahai, A., Machiraju, V., Ouyang, J., and Wurster, K. Message tracking in SOAP-based Web services. In *Network Operations and Management Symposium 2002 (NOMS '02)*, pages 33–47, Florence, Italy. IEEE, 2002.
- [Salehie et al., 2009] Salehie, M., Li, S., and Tahvildari, L. Employing aspect composition in adaptive software systems: a case study.

Bibliography

In *1st Workshop on Linking Aspect Technology and Evolution*, pages 17–21, Charlottesville, VA, USA. ACM, 2009.

- [Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4:14:1–14:42, 2009.
- [Salfner et al., 2010] Salfner, F., Lenk, M., and Malek, M. A survey of online failure prediction methods. *ACM Computing Surveys*, 42:10:1–10:42, 2010.
- [Sauer et al., 1980] Sauer, C. H., MacNair, E. A., and Salza, S. A language for extended queuing network models. *IBM Journal of Research and Development*, 24(6):747–755, 1980.
- [Scheer, 2000] Scheer, A.-W. *ARIS – Business Process Modeling*. Springer, 3rd edition, 2000.
- [Schneider et al., 2007] Schneider, F. T., Payer, M., and Gross, T. R. Online optimizations driven by hardware performance monitoring. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 373–382, San Diego, CA, USA. ACM, 2007.
- [Shumway and Stoffer, 2006] Shumway, R. H. and Stoffer, D. S. *Time Series Analysis and its Applications: With R Examples*. Springer, 2006.
- [Smith and Williams, 2002] Smith, C. U. and Williams, L. G. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison Wesley, 2002.
- [Smith and Williams, 2003] Smith, C. U. and Williams, L. G. Best practices for software performance engineering. In *Proceedings of the 29th International Conference of the Computer Measurement Group on Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG 2003)*, pages 83–92, Dallas, TX, USA. Computer Measurement Group, 2003.

Bibliography

- [Snatzke, 2009] Snatzke, R. G. Performance survey 2008 – survey by codecentric GmbH. <http://www.codecentric.de/export/sites/homepage/...resources/pdf/studien/performance-studie.pdf>, 2009.
- [Sosnoski, 2004] Sosnoski, D. Java programming dynamics, Part 7: Bytecode engineering with BCEL. *IBM developerWorks*, 2004.
- [Spencer, 2000] Spencer, J. Architecture description markup language (ADML): Creating an open market for it architecture tools. The Open Group, White Paper, <http://www.opengroup.org/tech/architecture/adml/background.htm>, 2000.
- [Spring Source, 2011] Spring Source. Spring Java Application Framework 3.1 – reference documentation. <http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/pdf/spring-framework-reference.pdf>, 2011.
- [Steinberg et al., 2008] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2008.
- [Steinder and Sethi, 2004] Steinder, M. and Sethi, A. S. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165–194, 2004.
- [Stewart, 2009] Stewart, W. J. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, 2009.
- [Streekmann and Hasselbring, 2008] Streekmann, N. and Hasselbring, W. Towards identification of migration increments to enable smooth migration. In *Model-Based Software and Data Integration: 1st International Workshop*, volume 8 of *Communications in Computer and Information Science*, pages 79–90, Berlin, Germany. Springer, 2008.
- [Sun Microsystems, Inc., 2004] Sun Microsystems, Inc. Java Remote Method Invocation Specification. <http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf>, 2004.

Bibliography

- [Sun Microsystems, Inc., 2006] Sun Microsystems, Inc. Java Management Extensions (JMX) Specification, Version 1.4. http://download.oracle.com/javase/7/docs/technotes/guides/jmx/JMX_1_4_specification.pdf, 2006.
- [Sun Microsystems, Inc., 2009] Sun Microsystems, Inc. Java Platform Enterprise Edition (Java EE) Specification, v6. <http://www.jcp.org/en/jsr/detail?id=316>, 2009.
- [Sundaresan et al., 2000] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.
- [Sweeney et al., 2004] Sweeney, P. F., Hauswirth, M., Cahoon, B., Cheng, P., Diwan, A., Grove, D., and Hind, M. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium (VM '04)*, pages 57–72, San Jose, CA, USA. USENIX, 2004.
- [Sydor, 2010] Sydor, M. J. *APM Best Practices: Realizing Application Performance Management*. Apress, 2010.
- [Systä et al., 2001] Systä, T., Koskimies, K., and Müller, H. Shimba – an environment for reverse engineering Java software systems. *Software – Practice & Experience*, 31:371–394, 2001.
- [Szyperski et al., 2002] Szyperski, C., Gruntz, D., and Murer, S. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 2nd edition, 2002.
- [Tanter et al., 2003] Tanter, E., Noyé, J., Caromel, D., and Cointe, P. Partial behavioral reflection: spatial and temporal selection of reification. *SIGPLAN Notices*, 38:27–46, 2003.
- [Taylor et al., 2009] Taylor, R. N., Medvidovic, N., and Dashofy, E. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

Bibliography

- [Teiniker et al., 2005] Teiniker, E., Schmoelzer, G., Faschingbauer, J., Kreiner, C., and Weiss, R. A hybrid component-based system development process. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 152–159, Porto, Portugal. IEEE Computer Society, 2005.
- [Thakkar et al., 2008] Thakkar, D., Hassan, A. E., Hamann, G., and Flora, P. A framework for measurement based performance modeling. In *Proceedings of the 7th International Workshop on Software and Performance (WOSP '08)*, pages 55–66, Princeton, NJ, USA. ACM, 2008.
- [The Open Group, 2009a] The Open Group. *ArchiMate 1.0 Specification*. Van Haren, 2009.
- [The Open Group, 2009b] The Open Group. *TOGAF Version 9*. Van Haren, 2009.
- [Theodoridis and Koutroumbas, 2008] Theodoridis, S. and Koutroumbas, K. *Pattern Recognition*. Academic Press, 4th edition, 2008.
- [Trofin and Murphy, 2008] Trofin, M. and Murphy, J. Static verification of component composition in contextual composition frameworks. *International Journal on Software Tools for Technology Transfer*, 10(3):247–261, 2008.
- [van Hoorn et al., 2011a] van Hoorn, A., Frey, S., Goerigk, W., Hasselbring, W., Knoche, H., Köster, S., Krause, H., Porembski, M., Stahl, T., Steinkamp, M., and Wittmüss, N. DynaMod project: Dynamic analysis for model-driven software modernization. In *Joint Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM '11) and the 5th International Workshop on Software Quality and Maintainability (SQM '11)*, volume 708, pages 12–13, Oldenburg, Germany, 2011.
- [van Hoorn et al., 2011b] van Hoorn, A., Knoche, H., Goerigk, W., and Hasselbring, W. Model-driven instrumentation for dynamic analysis of legacy software systems. In *Proceedings of the 13th*

Workshop Software-Reengineering (WSR '11), pages 26–27, Bad Honnef, Germany, 2011.

- [van Hoorn et al., 2009a] van Hoorn, A., Rohr, M., Gul, A., and Hasselbring, W. 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, Cape Town, South Africa. ACM, 2009.
- [van Hoorn et al., 2008] van Hoorn, A., Rohr, M., and Hasselbring, W. Generating probabilistic and intensity-varying workload for web-based software systems. In *Performance Evaluation – Metrics, Models and Benchmarks: Proceedings of the SPEC International Performance Evaluation Workshop 2008*, volume 5119 of *Lecture Notes in Computer Science*, pages 124–143. Springer, 2008.
- [van Hoorn et al., 2009b] van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., and Kieselhorst, D. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, 2009.
- [van Ommering et al., 2000] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.
- [W3C, 2010] W3C. XQuery 1.0: An XML Query Language (Second Edition) – W3C Recommendation 14 December 2010. <http://www.w3.org/TR/xquery/>, 2010.
- [Wei, 2005] Wei, W. W. S. *Time Series Analysis: Univariate and Multivariate Methods*. Addison Wesley, 2nd edition, 2005.
- [Weigend et al., 2011] Weigend, J., Siedersleben, J., and Adersberger, J. Dynamische Analyse mit dem Software-EKG. *Informatik-Spektrum*, 34(5):484–495, 2011.

Bibliography

- [Westermann et al., 2010] Westermann, D., Happe, J., Hauck, M., and Heupel, C. The Performance Cockpit approach: A framework for systematic performance evaluations. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA '10)*, pages 31–38, Lille, France. IEEE, 2010.
- [Wettel et al., 2011] Wettel, R., Lanza, M., and Robbes, R. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 551–560, Honolulu, HI, USA. ACM, 2011.
- [Woodside et al., 2007] Woodside, M., Franks, G., and Petriu, D. C. The future of software performance engineering. In *Future of Software Engineering 2007 (FOSE '07)*, pages 171–187, Minneapolis, MN, USA. IEEE Computer Society, 2007.
- [Woodside et al., 2005] Woodside, M., Petriu, D. C., Petriu, D. B., Shen, H., Israr, T., and Merseguer, J. Performance by unified model analysis (PUMA). In *Proceedings of the 5th International Workshop on Software and Performance (WOSP '05)*, pages 1–12, Palma, Spain. ACM, 2005.
- [Wooldridge, 2009] Wooldridge, M. J. *An Introduction to MultiAgent Systems*. Wiley, 2nd edition, 2009.
- [Xi et al., 2009] Xi, C., Harbulot, B., and Gurd, J. R. Aspect-oriented support for synchronization in parallel computing. In *Proceedings of the 1st Workshop on Linking Aspect Technology and Evolution (PLATE '09)*, pages 1–5. ACM, 2009.
- [Yilmaz et al., 2008] Yilmaz, C., Paradkar, A., and Williams, C. Time will tell: Fault localization using time spectra. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 81–90, Leipzig, Germany. ACM, 2008.
- [Zachman, 1987] Zachman, J. A. A framework for information systems architecture. *IBM Systems Journal*, 26(3):276–292, 1987.
- [Zeller, 2009] Zeller, A. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2nd edition, 2009.