Karlsruhe Institute of Technology

Architecture-Level Software Performance Models for Online Performance Prediction

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften/
Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation

von

Fabian Maria Konrad Brosig
aus Minden

Tag der mündlichen Prüfung:	10.07.2014
Erstgutachter:	Prof. Dr. Samuel Kounev
Zweitgutachter:	Prof. Dr. Ralf Reussner

Abstract

Modern enterprise systems often have distributed application architectures composed of many independent services running in a heterogeneous environment (Papazoglou et al., 2007). In such systems, applications are customized and new services are composed and deployed on-the-fly subjecting the system resources to varying workloads. Moreover, existing services, given their loosely-coupled nature, can evolve independently of one another. Managing system resources in such environments to ensure acceptable end-to-end performance and availability, while at the same time optimizing resource utilization, is a challenge (Armbrust et al., 2009; Brooks, 2011). Service providers are often faced with questions such as: What would be the performance of a given service if the workload continues to evolve as currently observed? What performance would a new service deployed on the infrastructure exhibit and how much resources should be allocated to it? How should the workloads of the running services be partitioned among the available resources such that performance objectives are met and resources are utilized efficiently? Answering such questions requires the ability to predict at *run-time*, i.e., during system operation, how the performance of running services would be affected if the workload or the system configuration changes. We refer to this as *online performance prediction*.

Existing approaches to online performance prediction (e.g., Nou et al. (2009); Li et al. (2009); Jung et al. (2010)) are based on *predictive* stochastic performance models such as (layered) Queueing Networks or Queueing Petri Nets. Such models normally abstract the system at the resource level without explicitly taking into account the software architecture. Services are typically modeled as black boxes, i.e., the control flow and the dependencies between software components are neglected. Detailed models that explicitly capture the software architecture and configuration exist in the literature (e.g., Becker et al. (2009); Grassi et al. (2007); Bertolino and Mirandola (2004)). They are typically software architecture models annotated with descriptions of the system's performance-relevant behavior. Such models, often referred to as *architecture-level* performance models, are intended to evaluate alternative system designs at design-time and/or predict the system performance for capacity planning purposes. However, there are fundamental differences between offline and online scenarios for performance prediction. This leads to different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques suitable for use at design-time versus run-time.

To realize proactive performance and resource management, overload situations should be anticipated and suitable reconfiguration actions must be found and triggered *before* Service Level Agreements (SLAs) are violated. In this context, there are situations where performance prediction results need to be available very fast to adapt the system *before* performance issues arise, as well as situations where a fine-grained prediction is needed to find an efficient system configuration. In this thesis, we develop novel architecture-level performance model abstractions for component-based software systems specifically designed for *online* use. We provide modeling and prediction facilities that enable online performance prediction during system operation. Performance questions can be answered

on the model level, i.e., analyses about the impact of workload changes or system reconfigurations can be conducted *without* executing expensive performance tests. Thus, our work provides a solid basis for developing model-based autonomic performance and resource management techniques that continuously adapt the system during operation in order to ensure that performance objectives are satisfied while at the same time system resources are used efficiently.

The major scientific contributions presented in this thesis are:

- **Novel architecture-level software performance abstractions for use in online scenarios.** This involves: (i) a new approach to model performance-relevant service behavior at different levels of granularity, (ii) a new approach to parameterize performance-relevant properties of software components, and (iii) a new approach to model dependencies between parameters, each specifically designed for use in online scenarios. The developed service behavior models are usable in different online performance prediction scenarios with different goals and constraints. They need to be parameterized with, e.g., resource demands, frequencies of external calls and branching probabilities. Given the online context, these parameters are characterized with probability distributions based on monitoring data collected at run-time. Furthermore, the behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation. In many practical situations, providing an explicit characterization of the dependency is not feasible and we thus introduce a suitable probabilistic representation based on monitoring data. The presented models enable online prediction of the performance impact of changing service compositions, resource allocation changes, system reconfigurations, and changing user behavior. The modeling abstractions are part of the Descartes Modeling Language (DML), a new modeling language for run-time performance and resource management of modern dynamic IT service infrastructures.

- **A tailored performance prediction process.** Online performance prediction needs to strike a balance between prediction accuracy and time-to-result. Accurate fine-grained performance predictions come at the cost of higher prediction overhead. By using more coarse-grained performance models one can speed up the prediction process. As a solution, we propose a performance prediction process that is *tailored* to a requested performance query: A performance query specifies which performance metrics should be predicted, e.g., which service response times need to be predicted, if response time percentiles are requested or average response times are sufficient. Furthermore, the performance query provides means to specify a trade-off between prediction accuracy and time-to-result, indicating if the query result needs to be available very fast at the expense of prediction accuracy or a longer prediction process with higher overhead is acceptable. Based on the performance query, the performance prediction process selects a suitable model solving technique and abstraction level, and returns the requested performance metrics. The tailored performance prediction process encapsulates complex domain knowledge on stochastic performance models and provides a flexible usage in different online performance prediction scenarios.

- **Methods for the integration of architecture-level performance models and system environments.** For online performance prediction and proactive system adaptation, it is essential to keep the performance model in sync with the modeled system. Otherwise, once a performance model of the system is built, the performance model may quickly become outdated and would thus not be representative of the real system anymore. Our approach is to *tie* the performance model to the system environment, i.e., to continuously adapt the model during system operation. We describe methods to ensure that the model is a "causally connected self-representation

of the associated system" (Blair et al., 2009) such that it constantly *mirrors* the performance-relevant structure and behavior of the system. The integration is realized by a method to extract model instances semi-automatically based on monitoring data and techniques to automatically maintain the extracted instances at run-time. Models are thus kept up-to-date and provide exact information about the system to enable accurate performance predictions.

The contributions are evaluated in two representative case studies. One case study uses the SPECjEnterprise2010 benchmark, it is a benchmark designed to serve as a representative application of today's enterprise Java systems. The second case study is a real-life enterprise software system from a large Software-as-a-Service (SaaS) provider. In the SPECjEnterprise2010 case study, the attained prediction accuracy in various realistically-sized deployment scenarios was below 5% error for resource utilization, and mostly within 10% to 20% and not exceeding 30% error for response time predictions which is considered acceptable for capacity management. The prediction capabilities were used as a basis for implementing an approach to autonomic performance-aware resource management. The effects of changes in user workloads as well as the impacts of reconfiguration actions were predicted with sufficient accuracy to avoid SLA violations or inefficient resource usage. In the SaaS provider case study the prediction accuracy is investigated under different workload types, different workload intensities and different workload mixes. The achieved prediction accuracy for resource utilization was within 5% error. For the service response times, the relative prediction error was mostly within 20%. The case studies demonstrate: i) that the proposed model abstractions lend themselves well to describe architecture-level performance models that are representative in terms of the performance properties of the modeled systems, ii) that the proposed prediction mechanisms can be effectively used to derive performance predictions in online scenarios, and iii) that the proposed model extraction and maintenance methods are suitable to extract and maintain performance model instances that provide an acceptable accuracy.

Our approach is the first approach to online performance prediction that uses architecture-level performance models. The performance prediction process facilitates flexible model-based predictions at run-time, combining the strength of simulative as well as analytical model solving techniques in a novel tailored process. The models are kept up-to-date using monitoring data, making manual error-prone parameter estimation unnecessary. The proposed approach offers a solid basis for implementing model-based autonomic performance and resource management techniques that continuously adapt the system during operation in order to ensure that performance objectives are met while efficiently using system resources. The vital benefit of employing models for system adaptation is that the performance models provide relevant information for what-if analyses and thus can drive the autonomic decision-making process. It is possible to search for valid and suitable system configurations on the model level and thus, unnecessary and possibly costly adaptations of the system can be avoided. This thesis opens up a range of new research possibilities. For example, the thesis of Huber (2014) is built directly on our approach, implementing a framework for autonomic performance-aware resource management. Other ongoing dissertation projects are going to use DML as a basis. Moreover, the work is already used in active collaborations with the industry applying it to real-life systems and applications.

Zusammenfassung

Nutzer erwarten von Geschäftsanwendungen schnelle Dienstantwortzeiten ohne Verzögerungen. Da heutige Geschäftsanwendungen anpassbar und erweiterbar konzipiert sind, haben sie oftmals komplexe Software-Architekturen, bestehend aus lose gekoppelten Software-Diensten (Papazoglou et al., 2007). Die Einhaltung der Performance-Anforderungen bei gleichzeitig effizienter Verwaltung der System-Ressourcen gilt daher als Herausforderung (Armbrust et al., 2009; Brooks, 2011). Dienstanbieter sind dabei mit Fragen konfrontiert, wie zum Beispiel: Wie verändern sich die Dienstantwortzeiten, wenn sich die Nutzerzahlen weiter dem beobachteten Trend nach entwickeln? Welche Antwortzeiten sind zu erwarten, wenn ein neuer Dienst auf der vorhandenen IT-Infrastruktur zur Verfügung gestellt wird? Wie viele System-Ressourcen müssen für den neuen Dienst alloziert werden? Wie müssen dabei das System umkonfiguriert bzw. die Ressourcenzuweisungen verändert werden, so dass vorhandene Ressourcen effizient genutzt werden? Um Fragen dieser Art beantworten zu können, muss die Performance des Software-Systems unter verschiedenen Konfigurationen und Auslastungsgraden zur Laufzeit vorhergesagt werden können. Für solche Performance-Vorhersagen werden Modelle (Performance-Modelle) benötigt, die relevante Performance-Eigenschaften des betrachteten Systems abbilden.

Existierende Ansätze für die Performance-Vorhersage zur Laufzeit (e.g., Nou et al. (2009); Li et al. (2009); Jung et al. (2010)) basieren auf *prädiktiven* stochastischen Performance-Modellen wie (mehrschichtige) Warteschlangennetzwerke oder Warteschlangen-Petri-Netze. Solche Modelle abstrahieren das System normalerweise auf Ressourcenebene, ohne die Software-Architektur explizit zu berücksichtigen. Dienste werden häufig als "Black Box" modelliert, Kontrollflüsse und Abhängigkeiten zwischen Software-Komponenten werden also nicht beachtet. Detaillierte Performance-Modelle, die die Software-Architektur und die Systemkonfiguration abbilden, existieren in der Forschungsliteratur (e.g., Becker et al. (2009); Grassi et al. (2007); Bertolino and Mirandola (2004)). Es sind typischerweise Software-Architekturmodelle, die zusätzlich mit Beschreibungen des Performance-relevanten Systemverhaltens annotiert werden. Solche Modelle, auch als *deskriptive* Performance-Modelle auf Architekturebene bezeichnet, wurden mit dem Ziel entwickelt, verschiedene System-Entwurfsalternativen noch während der Entwurfszeit hinsichtlich ihrer Performance-Eigenschaften gegeneinander abzuwägen. Es gibt jedoch fundamentale Unterschiede zwischen den Performance-Vorhersageszenarien zur Entwurfszeit und zur Laufzeit. Diese Unterschiede führen zu unterschiedlichen Anforderungen an die zugrundeliegenden Modellabstraktionen und zugehörigen Vorhersagetechniken für die Performance-Vorhersage jeweilig zur Entwurfszeit und zur Laufzeit.

Ein pro-aktives Performance- und Ressourcenmanagement zeichnet sich dadurch aus, das Überlast-Situationen antizipiert werden und daraufhin geeignete Rekonfigurationsaktionen gefunden und ausgeführt werden, noch bevor eventuelle Vereinbarungen zur Dienstgüte, auch Service Level Agreements (SLAs) genannt, verletzt sind. In diesem Kontext gibt es Situationen, in denen Performance-Vorhersagen sehr schnell verfügbar sein müssen um das System anpassen zu können bevor Performance-Probleme auftreten. Es gibt aber auch Situationen, in denen eine feingranulare Vorhersage vonnöten ist, um besonders effiziente

Systemkonfigurationen ermitteln zu können. In dieser Arbeit werden neue Performance-Modellabstraktionen auf Architekturebene speziell für die Performance-Vorhersage zur Laufzeit entwickelt. Performance-Anfragen können dadurch zur System-Laufzeit auf Modellebene beantwortet werden, d.h. ohne teure Performance-Tests ausführen zu müssen. Damit ist unsere Arbeit eine geeignete Grundlage für die Entwicklung von Mechanismen für modellbasiertes autonomes Performance- und Ressourcenmanagement, die das System kontinuierlich während seines Betriebs mit dem Ziel anpassen, die Performance-Anforderungen unter effizienter Verwendung der zur Verfügung stehenden Ressourcen einzuhalten.

Die wissenschaftlichen Kernbeiträge dieser Arbeit sind:

- **Neue Performance-Abstraktionen auf Architekturebene für die Verwendung zur Performance-Vorhersage zur Laufzeit.** Diese beinhalten: (i) einen neuen Ansatz zur Modellierung der Performance-relevanten Eigenschaften eines Dienstes in verschiedenen Granularitätsstufen, (ii) einen neuen Ansatz für die Parametrisierung der Performance-relevanten Eigenschaften von Software-Komponenten, und (iii) einen neuen Ansatz für die Modellierung von Abhängigkeiten zwischen Parametern, jeweils spezifisch für die Performance-Vorhersage zur Laufzeit entworfen. Die Modelle zur Dienstbeschreibung sind nutzbar in verschiedenen Performance-Vorhersageszenarien zur Laufzeit, die sich in Ziel und Randbedingungen unterscheiden können. Die vielfältigen Granularitätsstufen tragen dabei auch der oft unterschiedlichen Verfügbarkeit von Messdaten Rechnung. Die Dienstbeschreibungen werden u.a. mit Angaben zu Ressourcenbedarfen, Aufrufhäufigkeiten von externen Diensten und Verzweigungswahrscheinlichkeiten parametrisiert. Dies geschieht auf der Basis von Messdaten, die während der Laufzeit erhoben werden. Weiterhin führen wir die Möglichkeit ein, Abhängigkeiten zwischen Modellparametern probabilistisch, ebenso auf der Basis von Messdaten zu charakterisieren. Eine explizite funktionale Beschreibung der Abhängigkeiten kann nämlich oftmals nicht bereitgestellt werden. Die vorgestellten Modelle ermöglichen es, den Performance-Einfluss von Veränderungen der Dienstkomposition, von Veränderungen der Ressourcenallokation und Systemkonfiguration, und von Veränderungen des Nutzungsverhaltens zur Laufzeit vorherzusagen. Die Modellabstraktionen sind Bestandteil der Descartes Modeling Language (DML), einer neuen Modellierungssprache für das Performance- und Ressourcenmanagement von modernen IT Infrastrukturen.

- **Ein flexibles Verfahren zur Performance-Vorhersage.** Performance-Vorhersagen zur Laufzeit müssen eine geeignete Balance zwischen Vorhersagegenauigkeit und Vorhersagegeschwindigkeit finden. Genaue, feingranulare Performance-Vorhersagen bedeuten einen hohen Vorhersage-Overhead, wohingegen grobgranulare Performance-Modelle schnelle Vorhersagen liefern können. Je nach Situation kann es erforderlich sein, auf Vorhersagegenauigkeit zu verzichten um die Vorhersagegeschwindigkeit zu erhöhen. Wir schlagen ein Verfahren zur Performance-Vorhersage vor, das auf anstehende Performance-Anfragen jeweils flexibel zugeschnitten wird. So eine Performance-Anfrage spezifiziert, welche Performance-Metriken wie z.B. durchschnittliche Antwortzeit oder Ressourcenauslastung ermittelt werden sollen. Weiterhin beinhaltet eine Performance-Anfrage einen Hinweis, wie der Zielkonflikt zwischen Vorhersagegenauigkeit und Vorhersagedauer gewichtet werden soll, je nachdem ob zugunsten einer verkürzten Vorhersagedauer auf Vorhersagegenauigkeit verzichtet werden kann oder nicht. Ausgehend von einer Performance-Anfrage wird der Vorhersage-Prozess zugeschnitten. Je nach Anfrage werden geeignete Modell-Granularitätsstufen und Modell-Lösungsverfahren ausgewählt. Das zugeschnittene Vorhersageverfahren kapselt umfangreiches Domänenwissen über stochastische Performance-Modelle und bietet eine breite Nutzbarkeit in verschiedenen Vorhersageszenarien zur Laufzeit.

- **Methoden für die Integration von Performance-Modellen mit der Systemumgebung.** Für die Performance-Vorhersage zur Laufzeit ist es unabdingbar, dass das Performance-Modell mit dem modellierten System über die Zeit in Übereinstimmung gehalten wird. Andernfalls kann ein einmal erstelltes Modell schnell veralten und damit nur ungenaue Vorhersagen liefern. Unser Ansatz ist es, das Performance-Modell an die Systemumgebung zu binden, so dass es kontinuierlich während der Laufzeit mit dem System evolviert, und das System zu jeder Zeit widerspiegelt. Ermöglicht wird solch eine Integration durch eine Methode zur semi-automatischen Extraktion von Performance-Modellen aus Messdaten, und durch Techniken zur automatischen Wartung der Performance-Modelle zur Laufzeit. Die Performance-Modelle werden somit aktuell gehalten, und versprechen daher genaue Performance-Vorhersagen.

Die Beiträge dieser Arbeit werden in zwei repräsentativen Fallstudien evaluiert. Eine Fallstudie nutzt den SPECjEnterprise2010 Benchmark, der einer repräsentativen Geschäftsanwendung entspricht. Die zweite Fallstudie nutzt eine reale Geschäftsanwendung eines großen Software-as-a-Service (SaaS) Anbieters. In der SPECjEnterprise2010 Fallstudie wurde in verschiedenen Szenarien eine Vorhersagegenauigkeit von unter 5% Abweichung bezüglich der Ressourcenauslastung und unter 30% Abweichung (meist zwischen 10% bis 20%) bezüglich der vorhergesagten Antwortzeiten erreicht. Diese Abweichungen gelten für die Kapazitätsplanung als akzeptabel. Die Vorhersagen wurden zudem als Basis verwendet, um einen Ansatz zum autonomen Performance- und Ressourcenmanagement zu implementieren. Die Performance-Auswirkungen von Veränderungen im Nutzungsprofil sowie von Systemkonfigurationen konnten dabei mit einer Genauigkeit vorhergesagt werden, die zur Vermeidung von SLA Verletzungen oder ineffizienter Ressourcennutzung ausreichend waren. In der Fallstudie mit dem SaaS Anbieter wurden Performance-Vorhersagen unter verschiedenen Nutzungsarten und verschiedenen Nutzungsintensitäten untersucht. Die erreichte Vorhersagegenauigkeit bezüglich der Ressourcenauslastung lag im Bereich von 5% Abweichung. Für die Dienstantwortzeiten lag der relative Vorhersagefehler meist bei 20%. Die beiden Fallstudien zeigen, (i) dass die vorgeschlagenen Modellabstraktionen gut geeignet sind Performance-Modelle zu beschreiben, die die Performance-Eigenschaften eines Systems angemessen abbilden, (ii) dass die entwickelten Vorhersagemechanismen in der Lage sind, auf Basis der Performance-Modelle Vorhersagen zur Laufzeit abzuleiten, und (iii) dass die vorgeschlagenen Modellextraktions- und Modellwartungsmethoden dazu geeignet sind, Performance-Modelle von akzeptabler Genauigkeit zu erhalten.

Der vorgestellte neue Ansatz nutzt für die Performance-Vorhersage zur Laufzeit Modelle, die die Performance-relevanten Eigenschaften eines Systems auf Architekturebene beschreiben. Der vorgeschlagene Vorhersageprozess vereinfacht flexible Performance-Vorhersagen zur Laufzeit, indem er die Stärken von simulativen und analytischen Modell-Lösungsverfahren in einer jeweils zugeschnittenen Vorhersage kombiniert. Performance-Modelle werden auf der Basis von Messdaten während der Laufzeit aktuell gehalten, so dass manuelle, oft fehlerträchtige, Parameterschätzungen nicht vonnöten sind. Damit ist unsere Arbeit eine geeignete Grundlage für die Entwicklung von Mechanismen für modellbasiertes autonomes Performance- und Ressourcenmanagement, die das System kontinuierlich während seines Betriebs mit dem Ziel anpassen, die Performance-Anforderungen unter effizienter Verwendung der zur Verfügung stehenden Ressourcen einzuhalten. Der große Gewinn, hierfür Modelle einzusetzen, besteht darin, dass die Performance-Modelle Informationen für “Was wäre wenn”-Analysen liefern, und damit einen automatisierten Entscheidungsprozess für die Systemrekonfiguration möglich machen. Die systematische Suche nach gültigen und geeigneten Systemkonfigurationen kann auf der Modellebene erfolgen, und dadurch unnötiges und kostspieliges Ausprobieren vermeiden. Die vorliegende Arbeit eröffnet vielfältige neue Forschungsmöglichkeiten. Zum Beispiel baut die Arbeit von

Huber (2014) direkt auf unserem Ansatz auf, um ein Rahmenwerk für die Entwicklung von Techniken zum autonomen Performance- und Ressourcenmanagement bereitzustellen. Andere laufende Dissertationsprojekte nutzen ebenfalls die DML als Basis. Weiterhin werden Ansätze der Arbeit bereits in einigen Kollaborationen mit Industriepartnern genutzt, und auf reale Systeme und Anwendungen angewendet.

Acknowledgements

This thesis would not have been possible without the great support of many people. First of all, I would like to thank my advisor Prof. Samuel Kounev for giving me the opportunity to write this thesis as a member of the Descartes research group and for being a great supervisor and mentor. I would like to thank him for his guidance, his inspiring ideas and motivation when working on my thesis and our research papers. Without his support, advice and encouragement, this work would not have been possible. I would also like to thank Prof. Ralf Reussner for taking over the part of the second reviewer, and for hosting the Descartes research group at SDQ.

My special thanks go to my colleague and friend Dr.-Ing. Nikolaus Huber, for his help and advice and patience and distraction and motivation during the different phases of the work, for sharing an office with me, and for proofreading this thesis.

From the SE group at the University of Würzburg, and the SDQ group at KIT, I want to thank my current and former colleagues I had the pleasure to work with. In alphabetical order: Steffen Becker, Franz Brosch, Erik Burger, Axel Busch, Zoya Durdik, Michael Faber, Giovanni Falcone, Thomas Goldschmidt, Henning Groenda, Prof. Jürgen Wolff v. Gudenberg, Jens Happe, Lucia Happe, Michael Hauck, Christoph Heger, Robert Heinrich, Jörg Henß, Nikolas Herbst, Heinz Herrmann, Georg Hinkel, Oliver Hummel, Matthias Huber, Elena Kienhöfer, Joakim v. Kistowski, Benjamin Klatt, Fritz Kleemann, Anne Koziolek, Heiko Koziolek, Rouven Krebs, Klaus Krogmann, Max Kramer, Steffen Kruse, Michael Kuperberg, Martin Küster, Michael Langhammer, Philipp Merkle, Aleksandar Milenkoski, Christof Momm, Marco Nehmeier, Qais Noorshams, Fouad ben Nasr Omri, Michal Papez, Pierre Parrend, Chris Rathfelder, Andreas Rentschler, Tatiana Rhode, Vanessa Martin Rodriguez, Kiana Rostami, Piotr Rygielski, Kai Sachs, Viliam Simko, Simon Spinner, Johannes Stammel, Susanne Stenglin, Christian Stier, Misha Strittmatter, Mircea Trifu, Catia Trubiani, Robert Vaupel, Jürgen Walter, Alexander Wert, and Dennis Westermann. I was lucky to work in a very enjoyable and fruitful working atmosphere, you supported me with very constructive feedback in individual discussions, research meetings, and doctoral rounds.

Furthermore, I would like to thank Fabian Gorsler, Philipp Meier, Simon Spinner for supporting me as students as part of their diploma/master's theses, and Daniel Funke, Fabian Gorsler, Daniel Hoske, Evgeny Matershev, Philipp Meier, Christian Moldovan, Steffen Slavetinsky, Georgios Treskas, and Fabian Wiesner, for supporting me as research students in various projects.

Finally, I would like to thank my parents, my sister and my brother for their support and encouragement throughout the years, which made it all possible.

Contents

1. Introduction

1.1 Motivation

Modern enterprise systems are expected to provide their services in a responsive manner (Armbrust et al., 2009). The performance requirements of such systems are important, requiring an efficient management of the system resources (Woodside et al., 2007). According to Compuware (2008), performance is one of the most decisive factors for successful software systems. Since today's enterprise systems are typically designed to be highly customizable and extensible, they often have complex distributed application architectures composed of many independent services running in a heterogeneous environment (Papazoglou et al., 2007; Fowler, 2002). In such systems, applications are customized and new services are composed and deployed on-the-fly subjecting the system resources to varying workloads. Moreover, existing services, given their loosely-coupled nature, can evolve independently of one another. The increased flexibility gained through the adoption of, e.g., paradigms like service-oriented architecture or technologies like virtualization, comes at the cost of higher system complexity due to the complex interactions between the applications sharing the physical infrastructure as well as the introduced gap between physical and logical resource allocations. Managing system resources in such environments to ensure acceptable end-to-end performance and availability, while at the same time increasing resource utilization, is a challenge (Armbrust et al., 2009; Brooks, 2011). For instance, changes in the workload behavior of one application can affect the performance of other applications if they are sharing resources and services.

To maintain performance requirements and increase resource efficiency, systems should be continuously adapted to changes in their environment. For example, the amount of resources allocated to each service should be continuously adjusted to match the changing resource demands resulting from variations in the customer workloads. Service providers are often faced with questions such as: What would be the performance of a given service if the workload continues to evolve as currently observed? What performance would a new service deployed on the infrastructure exhibit and how much resources should be allocated to it? How should the workloads of the running services be partitioned among the available resources such that performance objectives are met and resources are utilized efficiently? If an application experiences a load spike or a change of its workload profile, how would this affect the system performance? Which parts of the system architecture would require additional resources? Answering such questions requires the ability to predict at *run-time*, i.e., during system operation, how the performance of running services would be affected if the workload or the system configuration changes. We refer to this as *online performance prediction.* Note that in the following, we use the expressions 'online', 'run-time', 'during operation' interchangeably.

To enable performance prediction we need an abstraction of the real system that incorporates performance-relevant behavior, i.e., a performance model. The model has to represent performance-relevant parts of the system by reflecting the abstract system structure (Balsamo et al., 2004). The software architecture, the flow of control and dependencies between services, must be taken into account in order to be able to answer questions such as the ones listed above. The goal of this thesis is to develop architecture-level performance models specifically designed for online use. We provide modeling and prediction facilities that enable online performance prediction during system operation. Performance questions can thus be answered on the model level, i.e., statements about the impact of workload changes or system reconfigurations can be made *without* conducting expensive performance tests that are typically infeasible during system operation. Such performance predictions can be leveraged, e.g., by autonomic resource management approaches that continuously adapt the system at run-time. Before we discuss the shortcomings of current work and present our approach we take a closer look at the problem we just described.

1.2 Problem Statement

Modern enterprise systems are usually built following the principles of component-based software architectures (Szyperski et al., 2002; Fowler, 2002). A software component is defined as a building block of a software system with contractually specified provided and required interfaces (Szyperski et al., 2002). In this thesis, we refer to the functionality provided by a component as *service.* Consequently, we refer to the methods of the provided interfaces of a component as provided services, and to the methods of the required interfaces of a component as required services or external services.

Managing system resources to ensure acceptable end-to-end performance and availability, while at the same time increasing resource utilization, is a challenge (Armbrust et al., 2009; Brooks, 2011). To address this challenge, online performance prediction facilities are required to anticipate performance problems before they occur and to predict the performance impact of system reconfigurations or possible adaptation actions. Since measurement experiments are impractical during system operation, performance predictions should be carried out on the model level, i.e., without conducting performance tests.

We formulate the following requirements on a performance prediction mechanism for a component-based software system:

- The prediction mechanism should support performance metrics such as average resource utilization, service response time and service throughput. The average resource utilization is of interest for processing resources (e.g., CPUs) as well as for software resources (e.g., thread pools). For a service response time, estimating the mean, variance and distribution should be supported. The distribution is used to derive percentiles such as the 90th percentile, indicating an expected response time level for 90% of the requests. Since "the 90% percentile response time is closer to what a user would perceive in reality" (Liu, 2009), such percentiles are common metrics to reflect end-user performance. Service throughput is of interest when analyzing closed workloads, i.e., if the system workload is defined by a number of concurrent users and their think times.
- The prediction mechanism should support predicting the performance impact of changing service compositions. Service compositions include deploying or removing

a service, replacing a service with another service implementation, and changes of how the services are connected to each other.

- The prediction mechanism should support predicting the performance impact of changes of resource allocations and system reconfigurations. Resource allocation is the assignment of resources to software components. System reconfigurations include adding or removing physical or virtual machines, and changing performance-influencing system parameters such as thread pool sizes.
- The prediction mechanism should support predicting the performance impact of different load-intensity levels and usage profiles. The load-intensity level is defined either by the inter-arrival time of user requests or by the number of concurrent users and their think times. The usage profile captures the services that are called, the order in which they are invoked, and the input parameters passed to them.

To be able to conduct performance predictions *online*, additional challenges arise that induce the following additional requirements:

- Online prediction scenarios differ in their requirements for prediction accuracy and speed. The trade-off between prediction accuracy and time-to-result should be configurable. An accurate fine-grained performance prediction comes at the cost of higher prediction overhead and a longer prediction process. By using more coarse-grained performance models one can speed up the prediction process.
- For online performance predictions, it is essential to keep the performance model in sync with the modeled system. The model should provide up-to-date information about the system to enable accurate performance predictions. The models@runtime community describes such a model as "causally connected self-representation of the associated system" (Blair et al., 2009), i.e., it constantly *mirrors* the performance-relevant structure and behavior of the system.
- The prediction accuracy must be adequate to support reasoning about software service compositions, resource allocations and system reconfigurations to increase resource efficiency. According to Menasce and Virgilio (2000), for capacity planning a prediction error of 30% concerning mean response times and 5% concerning resource utilization is considered acceptable.

In the following, we discuss the shortcomings of current approaches to performance prediction that motivate our work presented in this thesis.

1.3 Current Approaches

Existing approaches to online performance prediction (e.g., Bennani and Menascé (2005); Nou et al. (2009); Li et al. (2009); Jung et al. (2010)) are based on stochastic performance models such as (layered) Queueing Networks or Queueing Petri Nets. Such models, often referred to as *predictive* performance models, normally abstract the system at a high level without explicitly taking into account its software architecture and configuration. The impact of changing service compositions is not possible to predict since the connections between services are not reflected. Instead, services are typically modeled as black boxes and many restrictive assumptions are often imposed such as a single workload class, single-threaded components, or homogeneous servers. Service demands and request inter-arrival times are often limited to exponential distributions.

There are also pure black-box approaches that use monitoring data to infer mathematical models (Eskenazi et al., 2004; Elkhodary et al., 2010; Westermann et al., 2012; Gambi et al., 2013). Systematic measurements (Westermann et al., 2012) serve as input to derive interpolations based on statistical regression models (e.g., Eskenazi et al. (2004)), Kriging models (Gambi et al., 2013), or other machine learning-based approaches (e.g., Elkhodary et al. (2010)). However, all these models only interpolate or extrapolate the measurements, a representation of the system architecture is not extracted and thus predictions for service composition changes and system reconfiguration scenarios cannot be obtained.

Detailed models that explicitly capture the software architecture and configuration have been proposed, however, such models are intended for use at design-time (e.g., Becker et al. (2009); Grassi et al. (2007); Bertolino and Mirandola (2004); Object Management Group (OMG) (2006)). Models in this area are usually software architecture models (e.g., based on UML) annotated with descriptions of the system's performance-relevant behavior. Such models, often referred to as *architecture-level* performance models, are used at design-time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes.

While architecture-level performance models provide a powerful tool for performance prediction, they are typically expensive to build and solve, and provide limited support for flexible abstraction levels, which renders them impractical for use at run-time. Recent efforts in the area of component-based performance engineering have contributed a lot to facilitate model reusability (Koziolek, 2010), however, there is still much work to be done before they can be used for online performance prediction. The fundamental differences between offline and online scenarios for performance prediction lead to different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques. In particular, the type and amount of data available as a basis for model parameterization and calibration at system design-time versus run-time is different, calling for different abstraction levels. Furthermore, current approaches to modeling the component context in architecture-level performance models, in particular dependencies of the component behavior on model parameters, are not suitable for use at run-time since they do not provide enough flexibility in the way parameter dependencies can be expressed and resolved.

1.4 Approach and Contributions of this Thesis

In this thesis, we develop novel modeling abstractions for describing component-based software systems and their performance relevant behavior at the architecture-level, specifically designed for online use. We provide modeling and prediction facilities that enable online performance prediction during system operation. Performance questions can thus be answered on the model level, i.e., analyses about the impact of workload changes or system reconfigurations can be conducted *without* executing expensive performance tests. Performance problems can be anticipated *before* they occur and the impact of possible adaptation actions can be predicted. Hence, our work provides a solid basis for developing model-based autonomic performance and resource management techniques that continuously adapt the system during operation in order to ensure that performance objectives are satisfied while at the same time system resources are used efficiently (Kounev et al., 2010).

1.4.1 Success Criteria

With the aim of being applicable and usable in real-world scenarios, our approach should fulfill the following success criteria that we consider essential for a modeling and prediction approach (cf. Rathfelder (2013)).

- Expressiveness: The approach should be applicable to software systems that have a component-based software architecture. The modeling and prediction capabilities should support the scenarios described in Section 1.2.

- Accuracy: The modeling and prediction techniques should provide sufficiently accurate results compared to the actual system's performance. Normally, deviations within 30% for response time and deviations within 5% for resource utilization are considered acceptable for capacity planning (Menasce and Virgilio, 2000).
- Scalability: The approach should support the modeling and evaluation of systems of realistic size and complexity.
- Automation: The approach should allow for a high degree of automation, meaning that it should be possible to automate most activities using tools in order to drive decision making as part of an autonomic system adaptation process.

1.4.2 Contributions

The contributions presented in this thesis are described in the following.

Novel Architecture-Level Performance Abstractions for Online Use

In this thesis, we propose novel architecture-level performance abstractions for online use. This involves: (i) a new approach to model performance-relevant service behavior at different levels of granularity, (ii) a new approach to parameterize performance-relevant properties of software components, and (iii) a new approach to model dependencies between parameters, each specifically designed for use at run-time.

It is important to support the modeling of service behavior at different levels of abstraction and detail because the models should be usable in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to accurate performance prediction. Moreover, the information that monitoring tools can obtain at run-time, e.g., to what extent component-internal monitoring data is available, differs. This needs to be reflected by supporting different model abstraction levels. We provide three service behavior abstractions. A "black-box" abstraction offers a probabilistic representation of the service response time behavior. This representation captures the view of the service behavior from the perspective of a service consumer without any additional information about the service's behavior. A "coarse-grained" abstraction captures the service behavior when observed from the outside at the component's boundaries. This abstraction requires information about the service's total resource consumption as well as information about external calls made by the service. However, no information about the service's internal control flow is required. A "fine-grained" abstraction captures the service's performance-relevant control flow, which is an abstraction of the actual control flow. In contrast to the coarse-grained behavior description, a fine-grained behavior description requires information about the internal service control flow including information about the resource consumption of component-internal actions.

The above-mentioned service behavior abstractions have to be parameterized with, e.g., resource demands, frequencies of external calls, and branching probabilities. In the context of online performance models, these parameters are typically characterized with probability distributions based on monitoring data collected at run-time. Our modeling abstractions allow specifying a so-called *scope* for model parameters that indicate if and how monitoring data for a given parameter, collected at the component instance level, can be aggregated with data collected at other component instances.

Furthermore, the behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation (Fowler, 2002; Rolia and Vetland, 1995). Such parameters are often not traceable directly over the service interface and tracing them requires looking beyond the component boundaries, e.g., the parameters might be passed to another component in the call path and/or they might be stored in a database structure queried by the invoked service. Moreover, the behavior of component services may be dependent on the state of data containers such as caches or on persistent data stored in a database (Fowler, 2002; Rolia and Vetland, 1995). In many practical situations, providing an explicit characterization of such parameter dependencies is not feasible and we thus introduce a suitable probabilistic representation based on monitoring data.

All the mentioned modeling abstractions are part of the Descartes Modeling Language (DML), a new modeling language for run-time performance and resource management of modern dynamic IT service infrastructures. The contributions presented above were published in Brosig et al. (2013b, 2012); Brosig (2011).

Tailored Performance Prediction Process

Techniques for proactive online performance and resource management aim at adapting the system configuration and resource allocations dynamically. Performance problems such as overload situations should be anticipated; and suitable reconfigurations should be found on the model level and triggered *before* Service Level Agreements (SLAs) are violated. In such a context, there are situations where the prediction results need to be available very fast to adapt the system *before* performance issues arise, as well as situations where fine-grained predictions are needed to find a suitable system configuration. However, accurate fine-grained performance predictions come at the cost of higher prediction overhead and a longer prediction process. By using more coarse-grained performance models one can speed up the prediction process. Online performance prediction needs to strike a balance between prediction accuracy and time-to-result.

In this thesis, we provide a performance prediction process that is *tailored* to a requested performance query. Performance queries are formulated using our proposed language named Descartes Query Language (DQL). DQL has means to express the demanded performance metrics for prediction as well as the goals and constraints in a specific prediction scenario. A performance query describes which performance metrics should be predicted, e.g., if resource utilization or service response times need to be predicted, if response time percentiles are requested or average response times are sufficient. Furthermore, the performance query provides means to specify a trade-off between prediction accuracy and time-to-result, indicating if the query result needs to be available very fast at the expense of prediction accuracy or if a longer prediction process with higher overhead is acceptable. To define the notion of a performance query, DQL provides a declarative interface to performance prediction techniques that simplifies and automates the process of using architecture-level software performance models for performance analysis.

Based on a given performance query, the performance prediction process selects a suitable model abstraction level and model solving technique, and returns the requested performance metrics. The prediction process uses existing model solving techniques based on established stochastic modeling formalisms. The process decides which concrete model solving technique to apply, and it also selects suitable configuration options of the applied model solving technique with the goal of tailoring the solution method to the specific performance query. Therefore, for each model solving technique and its configuration options, it is important to understand how it affects the performance prediction in terms of specific predictable metrics, prediction accuracy, and prediction overhead.

DQL has been developed in the master's thesis of Gorsler (2013) and published in Gorsler et al. (2014, 2013). The investigation of the trade-off between prediction accuracy and time-to-result is published in Brosig et al. (2014).

Methods for the Integration of Architecture-Level Performance Models and System Environments

For online performance prediction and proactive system adaptation, it is essential to keep the performance model in sync with the modeled system. Otherwise, once a performance model of the system is built, the performance model may quickly become outdated and would thus not be representative of the real system anymore.

Our approach is to *tie* the performance model to the system environment, i.e., to continuously adapt the model during system operation. In this thesis, we describe methods to ensure that the model is a "causally connected self-representation of the associated system" (Blair et al., 2009) such that it constantly *mirrors* the performance-relevant structure and behavior of the system. Models are thus kept up-to-date and provide exact information about the system to enable accurate performance predictions. As part of this thesis, we propose methods to integrate architecture-level performance models and system environments. The integration is realized by: (i) a technique to extract model instances semi-automatically based on monitoring data, and (ii) a technique to automatically maintain the extracted instances at run-time. The latter includes an interface between the performance model instance and the monitoring data that encapsulates the monitoring infrastructure and provides a clear separation of concerns. In each case, we distinguish between static structural information about the system environment (e.g., involved component types) and dynamic parameters (e.g., resource demands) that are reflected in the models.

Our approach is in line with Woodside et al. (2007) where a convergence of performance monitoring, modeling and prediction as interrelated activities is advocated. As the system components are implemented and deployed in the target production environment, the proposed techniques obtain representative estimates of the various model parameters taking into account the real execution environment. Moreover, model parameters are continuously adjusted to iteratively refine their accuracy. Performance-relevant information is monitored and described at the component instance level and not only at the component type level as typical for performance models at design-time. During operation, there is no possibility to run arbitrary experiments since the system is in production and is used by real customers issuing requests. In such a setting, monitoring has to be handled with care, keeping the monitoring overhead within limits such that system operation is not disturbed.

The contributions discussed above were published in Brosig et al. (2011, 2009, 2013a).

1.4.3 Evaluation

The contributions are evaluated with regard to the success criteria formulated in Section 1.4.1. In two case studies of two representative enterprise software systems, we show that the proposed architecture-level modeling abstractions are suitable to model the performance-relevant behavior of component-based software systems in an online context. The case studies are of realistic size and complexity. One case study uses the SPECjEnterprise2010 benchmark. It is a benchmark designed to serve as a representative application of today's enterprise Java systems. The other case study is a real-life enterprise software system from a large Software-as-a-Service (SaaS) provider.

In the SPECjEnterprise2010 case study, the attained prediction accuracy in various realistically-sized deployment environments under different workload mixes and load intensities was within 5% error for resource utilization, and mostly within 10% to 20% and not exceeding 30% error for response time predictions which is considered acceptable for capacity management. The prediction capabilities were used as a basis for implementing an autonomic performance-aware resource management technique. The effects of changes in user workloads as well as the impacts of reconfiguration actions could be predicted with sufficient accuracy to avoid SLA violations and inefficient resource usage. In the case study with the SaaS provider, the attained prediction accuracy for resource utilization was within 5% error. For service response times, the relative prediction error was mostly within 20%. This applies both to average service response times as well as to the 90th percentile response times, i.e., the response time distributions are also captured in a representative way.

With these case studies, we demonstrated: i) that the proposed performance abstractions lend themselves well to describe architecture-level performance models that are representative in terms of the performance properties of the modeled systems, ii) that the proposed prediction mechanisms can be effectively used to derive performance predictions from the performance models, and iii) that the proposed model extraction and maintenance methods are suitable to extract and maintain performance model instances that provide an acceptable accuracy.

The results of the evaluation were published in Brosig et al. (2013b, 2012, 2011); Huber et al. (2011a). The SaaS provider case study has not been published yet.

1.5 Application Scenarios

The developed performance modeling and prediction approach has been designed to be applicable in different scenarios. In this section, we provide an overview of possible application areas. Starting with separate scenarios for online capacity planning and impact prediction for workload changes and system adaptations, we present the application area of autonomic resource management, which is the most sophisticated area since it includes all of the previously mentioned scenarios.

Online Capacity Planning

Enterprise software systems should be scalable and provide the flexibility to handle different workloads. Classical performance analysis would require costly and time-consuming load testing for evaluating the system performance in different deployments. The developed modeling and prediction approach presented in this thesis enables performance engineers and system administrators to evaluate the system performance in heterogeneous hardware environments and to compare different deployment sizes in terms of their performance and efficiency. Given that model parameters are characterized using representative monitoring data collected at run-time, the prediction results exhibit higher accuracy than predictions obtained through design-time modeling approaches. The developed techniques help to answer the following questions that arise frequently during operation:

- What would be the average utilization of system components and the average service response times for a given workload and deployment scenario?
- How many servers are needed to ensure adequate performance under the expected workload?
- How much would the system performance improve if a given server is upgraded?

Impact Analysis of Workload Changes

In general, the workload intensity of enterprise software systems varies over time. The workload intensity may follow certain trends or patterns, e.g., a weekly pattern with low intensity over the weekend. In addition, there can be situations where it is foreseeable that the workload will double within the next month. Using workload forecasting approaches developed in our group in Herbst et al. (2014, 2013), it is possible to forecast future workload intensity trends. Based on the latter, our approach allows performance engineers and system administrators to anticipate performance problems. System behavior and performance can be easily evaluated for different workloads. In contrast to performance tests, the model-based approach allows evaluating the system without setting up a representative testbed. The prediction process allows both determining the maximal system throughput as well as detecting potential bottlenecks. The questions that arise in this scenario and that can be answered by applying the modeling and prediction techniques developed in this thesis are:

- What maximum load level can the system sustain for a given resource allocation?
- How does the system behave for the anticipated workload behavior?
- Which component or resource is a potential bottleneck for a certain workload scenario?

Impact Analysis of Service Recompositions and Reconfigurations as well as System Adaptations

Today's enterprise software systems running on modern application platforms allow performing comprehensive online reconfigurations and adaptations, without service disruption. Applications can be customized, new services can be composed and deployed on-the-fly, service configuration parameters can be changed. To provide an illustrative example, assume the default setting of the rowsPerPage parameter of a frequently accessed list view is changed, e.g., doubled from 25 to 50. The impact of such a reconfiguration may have a severe impact on the database server or application server utilization and/or a significant influence on the end-to-end service response times. With our approach to capturing probabilistic parameter dependencies, the impact of such a reconfiguration can be assessed in advance without conducting performance tests in a representative testbed. Questions that can be answered using the approach presented in this thesis are:

- How does the system behave if a new service is deployed?
- What is the performance impact of changing a certain configuration parameter?
- Does a service re-composition improve the perceived service response time?
- What would be the performance impact of changing a third party external service provider?

Autonomic Resource Management at Run-time

The concepts and methods presented in this thesis provide a solid basis for developing model-based *autonomic* performance and resource management techniques that proactively adapt the system to dynamic changes at run-time with the goal to satisfy performance objectives while at the same time ensuring efficient resource utilization.

State-of-the-art industrial mechanisms for automated performance and resource management generally follow a trigger-based approach when it comes to enforcing application-level SLAs concerning availability or responsiveness. Custom triggers can be configured that fire in a reactive manner when an observed metric reaches a certain threshold (e.g., high server utilization or long service response times) and execute certain predefined reconfiguration actions until a given stopping criterion is fulfilled (e.g., response times drop) (VMware, 2006; Amazon Web Services, 2010). However, application-level metrics, such as availability and responsiveness, normally exhibit a highly non-linear behavior on system load and they typically depend on the behavior of multiple servers across several application tiers. Hence, it is hard to determine general thresholds of when triggers should be fired given that the appropriate triggering points are typically highly dependent on the architecture of the hosted services and their workload profiles, which can change frequently during operation. The inability to anticipate and predict the effect of dynamic changes in the environment, as well as to predict the effect of possible adaptation actions, renders conventional trigger-based approaches unable to reliably enforce SLAs in an efficient and proactive fashion.

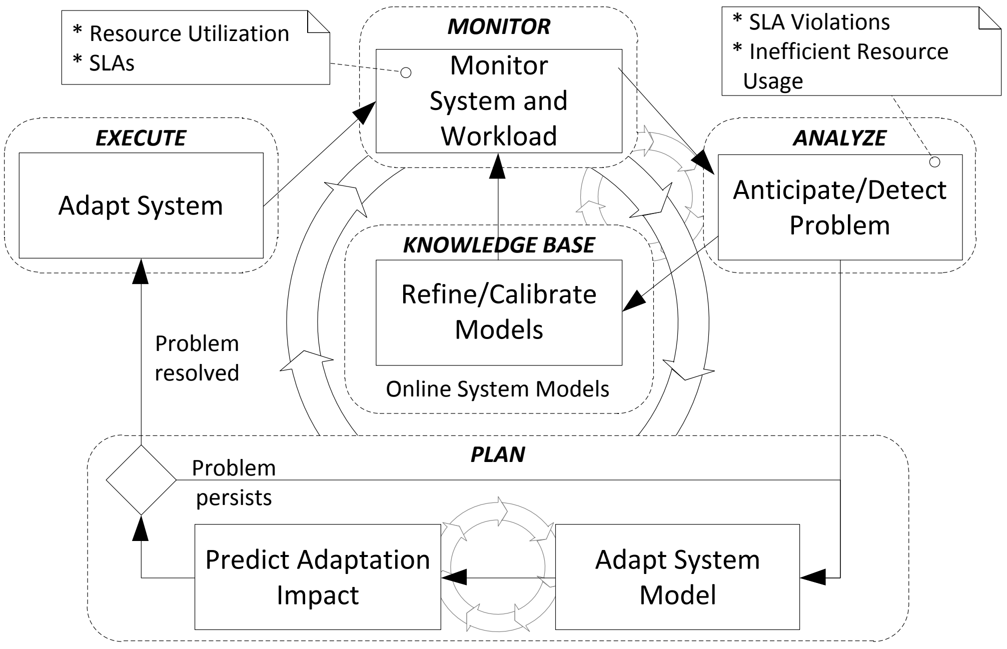


Figure 1.1: Model-Based System Adaptation Control Loop (Huber, 2014)

To overcome the mentioned shortcomings of current industrial approaches, Huber applied the methods presented in this thesis to develop a framework for autonomic performance-aware resource management (Huber, 2014). Figure 1.1 shows the control loop that is central to that framework. It consists of four main phases *Monitor*, *Analyze*, *Plan* and *Execute*. In addition, the figure depicts a *Knowledge Base* that is used by all mentioned phases. The knowledge base is realized with DML. Our work is used to conduct performance predictions on the model level to anticipate performance problems and to find suitable adaptation actions. DQL serves as interface between the management framework and our prediction facilities. Given that our performance prediction facilities support detailed impact analyses, e.g., workload intensity and usage profile changes, service (re-)compositions or deployment changes, the adaptation mechanisms can quickly converge to an efficient target system configuration (Huber et al., 2013). The tailored prediction process allows the adaptation mechanism to trigger predictions for multiple different configuration scenarios within a controllable period of time. The prediction results are sufficiently accurate since the models are maintained up-to-date based on representative monitoring data obtained at run-time.

Huber (2014) evaluates the framework end-to-end in two different representative case studies (beyond the ones considered in this thesis), demonstrating that it can provide significant efficiency gains of up to 50% without sacrificing performance guarantees, and that it is able to trade-off performance requirements of different customers in heterogeneous hardware environments. Furthermore, it is shown that the approach enables proactive system adaptation, reducing the amount of SLA violations by 60% compared to a trigger-based approach. The results of the case studies in Huber (2014) show that it is possible to apply architecture-level performance models and online performance prediction to perform autonomic system adaptation on the model level such that the system's operational goals are maintained. Different adaptation possibilities can be assessed without having to change the actual system.

1.6 Thesis Organization

The thesis is structured as follows.

In Chapter 2, we present the foundations. It describes different existing approaches to performance prediction that differ in the employed abstraction levels, the phases of the system lifecycle they are targeting, and the input parameter space they consider when conducting performance predictions. In particular, we describe predictive performance models based on stochastic analysis and simulation techniques, and descriptive architecture-level performance models. Furthermore, we provide an overview of monitoring approaches currently used in industry and academia to collect performance-relevant measurements at run-time.

Chapter 3 reviews related work, focusing on the research area of online performance prediction. Moreover, the chapter also reviews existing approaches to performance model extraction and maintenance.

Chapter 4 presents the proposed novel architecture-level performance modeling abstractions for online use. The proposed performance abstractions reflect the application architecture, the deployment of services in a resource landscape, as well as the application's usage profile, thus making it possible to predict the impact of workload changes, software adaptations, or changing resource allocations.

Chapter 5 describes how to conduct online performance predictions using the performance modeling abstractions described in Chapter 4. It presents the detailed concepts of the tailored prediction process. This involves a model composition step, parameterization and resolution of relationships between parameters as well as the actual model solving step. Furthermore, the notion of a performance query is formalized in this chapter.

Chapter 6 presents methods to integrate architecture-level performance models and system environments with the goal to keep the models up-to-date as the system evolves during operation. The monitoring capabilities that are prerequisites to use those methods are formulated. We present the semi-automatic extraction of architecture-level performance models based on system request tracing, model structure maintenance in the context of an autonomic resource management process, as well as the derivation of model parameter values. Moreover, it is discussed how architecture-level performance models can be calibrated and adjusted at run-time with the goal to increase their accuracy.

Chapter 7 first presents evaluation goals covering both the modeling and prediction capabilities of our approach. Then, the chapter describes two representative case studies. The first case study is a real-life enterprise software system from a SaaS provider. The second case study uses the SPECjEnterprise2010 benchmark. The case studies demonstrate that: (i) the proposed performance abstractions lend themselves well to describe architecture-level performance models that are representative in terms of the performance properties of the modeled systems, (ii) that the proposed prediction mechanisms are capable of deriving performance predictions in online scenarios based on the performance model instances, and (iii) that the proposed model extraction and maintenance methods are suitable to extract and maintain performance model instances that provide an acceptable accuracy.

Finally, Chapter 8 concludes the thesis. It gives a summary of the thesis' scientific contributions and their benefits. Moreover, the chapter gives directions for future work.

2. From Design-Time to Online Performance Models

Performance is a critical factor for successful software projects (Glass, 1998). Although hardware speed is continuously increasing, software performance problems are common since software system complexity and size are also growing at a fast pace (Woodside et al., 2007). A widespread misconception is that performance problems can be addressed by simply throwing enough hardware at the system (Kounev, 2005), also denoted as "kill it with iron" approach (Smith and Williams, 2002). Adding additional hardware resources can only resolve a performance problem if the existing resources are not sufficient. If the performance problem stems from a software bottleneck that is inherent in the software design, additional hardware resources may mitigate the problem, but will not solve it.

To avoid performance problems, it is important to analyze the expected performance characteristics of systems during all phases of their life cycle. The discipline Software Performance Engineering (SPE) focuses on methods and tools to address this challenge. SPE is described as a "systematic, quantitative approach to the cost-effective development of software systems to meet performance requirements" (Smith and Williams, 2002). SPE helps to estimate the level of performance a system can achieve and provides recommendations to realize the optimal performance level (Menascé et al., 2004b).

In this work, the term performance is understood as the degree to which a software system meets its objectives for timeliness and the efficiency with which it achieves this (Smith, 2002). Timeliness is measured in terms of meeting response time or throughput requirements and scalability goals. Scalability is understood as the ability of the system to continue to meet its objectives for response time and throughput as the demand for the services it provides increases and resources are added (Smith, 2002). Hence, performance involves *both* timing behavior *as well as* resource efficiency. Minimizing the application's resource demands while keeping the application's timing behavior at the required level is of increasing importance, in particular for energy-efficient computing (Barroso and Hölzle, 2007; Murugesan, 2008).

To ensure that a software system meets its performance requirements, the ability to predict its performance under different configurations and workloads is highly valuable throughout the system life cycle. During the design phase, performance prediction helps to evaluate different design alternatives. At deployment time, it facilitates system sizing and capacity planning. During operation, predicting the effect of changes in the workload or in the system configuration is crucial for an efficient resource management.

An alternative to performance prediction can be an expert's educated guess, or conducting performance measurement experiments. For a measurement experiment, the system needs to be deployed in an environment that reflects the configuration of interest, and needs to be stressed with the workloads of interest. Such experiments, however, are normally very expensive and time-consuming and therefore often not economically viable. To enable performance prediction, we need an abstraction of the real system that incorporates performance-relevant data, i.e., a performance model. Based on such a model, performance analysis of the system can be carried out.

This chapter describes different existing approaches to performance prediction. They differ in the employed abstraction levels, the phases of the system lifecycle they are targeting, and the input parameter space they consider when conducting performance predictions.

- We start with *black-box* approaches that abstract the system at a very high level without knowing any details about the internal system structure (Section 2.1). They use monitoring data to infer mathematical models, for example, systematic measurements (Westermann et al., 2012) serve as input to derive interpolations based on statistical regression models (e.g., Eskenazi et al. (2004)), Kriging models (Gambi et al., 2013), or other learning-based approaches (e.g., Elkhodary et al. (2010)). However, all such models only interpolate or extrapolate the measurements, a representation of the system architecture is not extracted and thus predictions for service composition changes and system reconfiguration scenarios cannot be obtained.

- Next, we describe *predictive* stochastic performance models that capture the temporal system behavior using stochastic analysis methods (Section 2.2). Formalisms such as (layered) Queueing Networks or Queueing Petri Nets are used to abstract the system at a high level without explicitly taking into account its software architecture and configuration. It is impossible to predict the impact of system reconfigurations and service (re-)compositions as the connections between services are not reflected. Instead, services are typically modeled as black boxes and many restrictive assumptions are often imposed such as a single workload class, single-threaded components, homogeneous servers, or exponential request inter-arrival times and exponential service demands.

- Finally, we describe *descriptive architecture-level* performance models that reflect the architectural structure of a software system and its performance-influencing factors (Section 2.3). They are detailed models that explicitly capture the software architecture and configuration, however, such models are intended for use at design-time (e.g., Becker et al. (2009); Grassi et al. (2007); Bertolino and Mirandola (2004); Object Management Group (OMG) (2006)). Models in this area are descriptive, i.e., software architecture models (e.g., based on UML) annotated with descriptions of the system's performance-relevant behavior.

Architecture-level performance models provide a powerful tool for performance prediction; they are used at design-time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes. However, in Section 2.4 we argue that there are fundamental differences between offline and online scenarios for performance prediction. This leads to different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques suitable for use at design-time versus run-time. Section 2.4 summarizes the main differences in terms of goals and underlying assumptions driving the evolution of online models.

Moreover, in Section 2.5 we provide an overview of monitoring approaches that are currently used in industry and academia to collect performance-relevant measurements at run-time.

2.1 Black-Box Approaches to Performance Prediction

Black-box approaches to performance prediction use measurements to infer mathematical models of the performance behavior. They abstract the system at a very high level without taking the structure of the system into account. The mathematical models interpolate and extrapolate the measurements, but the behavior *behind* the measurements is not captured. Inferring functional relationships from measurements, i.e., statistical learning, is an established research area (Hastie et al., 2001). In this section, we focus on approaches to model functional relationships between workload or configuration parameters as independent variables, on the one hand, and performance metrics such as response time, throughput or resource utilization as dependent variables, on the other hand.

Following the categorization proposed by Westermann (2013), black-box approaches differ from each other in the following aspects: Some approaches derive the training data during operation, i.e., without controlling the independent parameters. Other approaches vary the independent parameters in a systematic fashion with the goal to representatively capture a large input space (e.g., Westermann et al. (2012)). In addition, some approaches consider the trade-off between the amount of training data and the accuracy of the inferred function. Furthermore, the approaches differ in their assumptions about the form of the actual functional dependency, e.g., whether there is a linear dependency or not. While Westermann et al. (2012); Nadeem et al. (2006); Elkhodary et al. (2010); Gambi et al. (2013) build black-box models intended to directly represent system performance behavior, Courtois and Woodside (2000); Eskenazi et al. (2004) infer black-box models that serve as input to an ad-hoc performance model.

Westermann et al. (2012) propose an automated, measurement-based inference approach that is based on various strategies for iterative experiment selection and function inference. The focus is on deriving functional dependencies "with the least possible amount of data" (Westermann et al., 2012). As statistical methods to obtain mathematical functions, they apply Classification and Regression Trees (CART), Multivariate Adaptive Regression Splines (MARS), Kriging models, and genetic programming. As methods to efficiently explore the input space, methods such as random breakdown, adaptive equidistant breakdown, and an adaptive random breakdown are applied. The methods are compared and evaluated in two case studies based on benchmarks from SAP and from the Standard Performance Evaluation Corporation (SPEC), respectively. The independent parameters are different workload parameters and JVM configuration parameters, the dependent parameter is the average service response time in the SAP case study and the average throughput in the SPEC benchmark case study. The authors assume a representative testbed to be available in order to apply the approach.

In (Nadeem et al., 2006), the goal is to predict application execution times in grid environments. The predictions are used to make decisions about the efficient usage of grid resources. The authors suggest a two-layered training phase to minimize the training effort. Prediction functions obtained on a single grid resource are extrapolated to other resources. As independent parameters, all performance-relevant input parameter values of the application are considered. As dependent parameter, the application execution time is observed. The number of experiments is reduced by a normalization of the grid resources, assuming that the application's performance behavior for the input parameter variations is similar on the different grid resources. A performance prediction is based on a lookup in the training data set searching for the nearest reference value. The approach is specific to grid environments, the training phase is executed offline.

In Elkhodary et al. (2010), a learning-based approach is used to implement a self-adaptive software system. Instead of using an explicit model, the approach intends to "learn" the impact of adaptation decisions concerning the system's goals and use this knowledge for future adaptation decisions. The knowledge is iteratively refined in learning cycles. The first learning cycle has to be performed offline, before the system's initial deployment. The system is either simulated or executed in offline mode, corresponding metrics are collected serving as initial training data to induce a preliminary model of the system's behavior. The learning itself aims to discover relationships between so-called features and application-level metrics. "A feature is a domain and platform independent method of representing a particular system capability" (Elkhodary et al., 2010). The definition of the set of features to be considered in the learning process requires an engineer's domain knowledge. Each relationship is represented as a function quantifying the impact of the feature on a given metric. The function is characterized using a learning algorithm. While the approach is not tied to a particular algorithm, the authors implement the machine learning M5 model tree algorithm (Jordan and Jacobs, 1993) that eliminates insignificant features automatically.

Gambi et al. (2013) apply Kriging models (also referred to as Gaussian process regression or Kolmogorov Wiener prediction) to predict system behavior as part of an autonomic cloud resource controller. A Kriging model serves as black-box model of the system that evolves over time. The independent parameters are workload and system configuration data, the dependent parameters are application-level performance metrics such as average response times. The authors describe the benefits of applying Kriging models as follows: "We propose Kriging models because [..] the controller can train them quickly enough to apply them online, they don't require that designers define their internal structure, they provide good accuracy with small training datasets, and they can estimate the confidence of their predictions" (Gambi et al., 2013). The approach is evaluated in a case study with a non-disclosed application based on Sun Grid Engine middleware.

Courtois and Woodside (2000) uses controlled experiments to derive training data, i.e., to automatically infer functions that serve as performance prediction models. The varied independent parameters are configuration and service input parameters of a software component's provided service. The considered dependent parameter is the service resource demand. A case study showed that linear regression may not be sufficient to model the performance behavior. Instead, the authors apply the MARS method to fit the observed complex performance behavior. For that purpose, the authors use an experiment automation tool. Furthermore, the authors provide an estimation for the accuracy of the derived function based on MARS. The also propose a heuristic approach how to select a minimum number of experiments in order to reach a target accuracy level of the inferred function. In summary, the work enables automated curve fitting of non-linear functions while considering the trade-off between the amount of training data and the desired accuracy of the mathematical model. A representative testbed has to be available to apply the approach.

In Eskenazi et al. (2004), the authors use statistical regression to obtain prediction models for a component's provided services. A prediction model is described as a function with a service's signature type (consisting of only performance-relevant parameters) as input and the service's resource demand as output. Via regression methods, a function is derived from resource demand measurements conducted while the specific component service is executed in a testbed in the context of representative use cases. The performance of a system request is then computed by composing individual services' performance functions according to the control flow. Due to the required testbed, "factors affecting the perceived performance of a software component like influences by external services" (Becker et al., 2007) are neglected in this approach. In addition, to ensure the measurements' validity, the testbed must be stable during the process.

All the mentioned black-box approaches use existing methods to obtain a function from inputs to outputs, so that the function can return outputs for previously unseen inputs.

This is called Supervised Learning (Mohri et al., 2012). We do not provide more details about the applied methods here and instead refer to Hastie et al. (2001) and Izenman (2009).

2.2 Predictive Stochastic Performance Models

In this section, we explain *predictive* stochastic performance models that capture the temporal system behavior using stochastic analysis methods. Given that the model solving approach presented in this thesis is based on predictive stochastic models, we explain such models in more detail.

A number of different performance modeling formalisms have been developed. We provide an overview of some of the most popular types of performance models. We start with classical Queueing Networks (QNs), briefly describe Stochastic Petri Nets (SPNs), and then introduce Queueing Petri Nets (QPNs) that can be understood as a merger of QNs and SPNs (Bause, 1993; Kounev, 2005). We also describe Layered Queueing Networks (LQNs) which are an extension of QNs, and Stochastic Process Algebras (SPAs). Note that, for brevity, we refrain from formal definitions in this section.

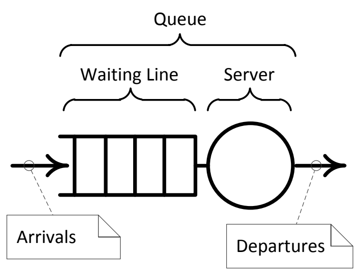
Queueing Networks (QNs)

QNs provide a powerful method for modeling contention for processing resources, i.e., hardware contention and scheduling strategies. The core entity of a QN is a queue (sometimes also denoted as service station). As depicted in Figure 2.1(a), a queue consists of a waiting line and a server. Requests arrive at the queue and are processed immediately at the server unless it is already occupied by another request. In the latter case, the request is put into the waiting line. If a request has been completely processed by the server, it departs from the queue. Figure 2.1(a) shows a single server queue. There are also queues consisting of multiple servers (denoted as multi-server queues), the semantics is similar: Whenever a request arrives it is processed at a server that is currently free. If all servers are occupied, the request is put into the waiting line.

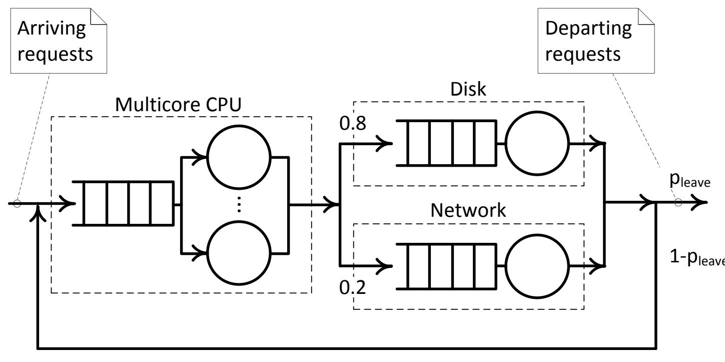
In the following, we introduce some common terms used in queueing theory (Menascé et al., 1994; Bolch et al., 1998; Trivedi, 2002). Requests may arrive at a queue at arbitrary points in time. The duration between successive request arrivals is denoted as *inter-arrival* time. The amount of requests per time unit is referred to as *arrival rate*, often denoted as λ. The time a server is occupied by a request is called *service time*. The time a request spends waiting in the waiting line is referred to as *queueing delay* or simply *waiting time*. The *response time* of a request is the total amount of time the request spends at the queue, i.e., the sum of waiting time and service time. Whenever a server has completed serving a request, another request waiting in the waiting line (if there is any) is scheduled with respect to a certain *scheduling strategy*. Typical scheduling strategies include First-Come-First-Served (FCFS), i.e., the first request in the waiting line is scheduled to be processed, Processor-Sharing (PS), i.e., all requests in the queue are scheduled according to an idealized round robin scheduling with an infinitely thin time slice, or Infinite-Server (IS), i.e., all requests in the queue are scheduled immediately as if the queue had an infinite number of servers. Scheduling strategy FCFS is typically used to model I/O devices, scheduling strategy PS is typically used to model CPUs, and scheduling strategy IS is typically used to model constant delays, e.g., average network delays.

There is a standard notation to describe a queue, referred to as *Kendall's notation* (Kendall, 1953). A queue is described by means of six parameters $A/S/m/B/K/SD$ (Kounev, 2005), where:

- A stands for the distribution of the inter-arrival time.



(a)



(b)

Figure 2.1: (a) Queue and (b) Queueing Network

- S stands for the distribution of the service time.
- m specifies the number of servers of the queue.
- B specifies the maximum number of requests that a queue can hold. If this parameter is missing, B is assumed to be infinite.
- K specifies the maximum number of requests that can arrive at the queue, i.e., K specifies the system population. If this parameter is missing, K is assumed to be infinite.
- SD stands for the scheduling strategy.

Typical notations for the distribution parameters are:

- M = Exponential (Markovian) distribution.
- D = Deterministic distribution, i.e. constant times without variance.
- E_k = Erlang distribution with parameter k.
- G = General distribution.

A deterministic distribution means that the respective times are constant. A general distribution means that the distribution is not known, e.g., this is commonly used for empirical distributions obtained from measurements if the underlying shape of the distribution is unknown. Parameters B and K are usually considered infinite and thus often omitted in a queue description.

Multiple queues can be connected to form a QN. Figure 2.1(b) shows an example with three queues, one multi-server queue and two single server queues. The multi-server queue represents a multicore CPU, the single server queues represent a disk and a network, respectively. The connections between the queues illustrate how requests are routed through the network of queues. An incoming request, after being processed by the CPU, is routed either to the disk or the network. The routes are labeled with probabilities. With a probability of 80 percent, a request coming from the CPU is routed to the disk queue. With a probability of 20 percent, a request coming from the CPU is routed to the queue representing the network. If a request is completed at either the disk or the network queue, the request either leaves the QN with a probability of p_{leave}, or it is immediately routed back to the CPU queue with a probability of $1 - p_{leave}$. Apparently, a request may visit a queue multiple times while circulating through the QN. A request's total amount of service time at a queue, added up over all visits of the queue, is called *service demand* or *resource demand* of the request at the queue. In the following, we use the term resource demand.

A QN where the requests come from a source that is external of the QN and leave the QN after service completion is referred to as *open* network. A QN where there is no such external source of requests and there are no departing requests, i.e., the population of requests in the QN remains constant and is equal to the initial population, is denoted as *closed* network. If a QN is open for some requests and closed for other requests, it is called *mixed*.

Given a QN, metrics of interest are, e.g., response time, *throughput*, i.e., the number of requests that are served per time unit, and utilization, i.e., the fraction of time where a queue is busy processing one or more requests. In order to analyze a QN quantitatively, the QN's *workload* needs to be specified. To distinguish different types of requests, requests are grouped into so-called *request classes* or *workload classes*. Requests of the same type are considered to behave similarly in terms of resource demand, population, inter-arrival time, and so on. For each workload class, the *workload intensity* as well as the *resource demands* for each visited queue have to be specified. How the workload intensity is characterized depends on whether it is a closed workload or an open workload. A closed workload is characterized by an (initial) number of requests, an open workload is characterized by an inter-arrival time of requests. A QN is said to be in *steady state* if the number of requests arriving at the QN per time unit is equal to the number of requests departing from the QN, i.e., the arrival rate is equal to the throughput. One very basic queueing theory law we use throughout this thesis is the Utilization Law (Menascé et al., 1994). Given a QN together with a workload specification, we can calculate the utilization $U_{i,r}$ of a queue/resource i due to requests of workload class r using the relationship

$$U_{i,r} = D_{i,r} \cdot X_{i,r}$$

where $D_{i,r}$ is the average resource demand and $X_{i,r}$ is the throughput of requests of workload class r at resource i. Closed formulas to derive response times of requests are not easy to derive, since they depend (amongst others) on the shape of the involved distributions, i.e., the inter-arrival time distribution, and resource demand distributions.

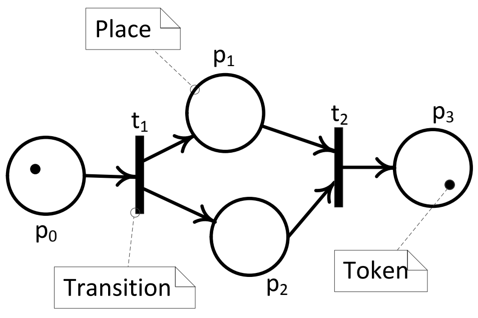
There is a special class of QNs, namely QNs that have a *product-form* solution. The BCMP theorem (of Baskett, Chandy, Muntz, Palacios) defines a set of conditions, i.e., a combination of service time distributions and scheduling strategies for which QNs with multiple workload classes can be efficiently solved using a product-form solution (Menascé et al., 1994; Bolch et al., 1998; Trivedi, 2002). For instance, if a queue has a FCFS scheduling strategy, then all service times at this queue need to be of the same exponential distribution. In general, QNs with efficient mathematical solutions are often based on strong assumptions (Jain, 1991). Common assumptions include exponentially distributed inter-arrival times and service time distributions. QNs with weaker assumptions are usually mathematically intractable. Thus, they need to be simulated in order to derive performance predictions.

Techniques to solve QNs are supported by, e.g., the SPEED tool (Smith and Williams, 1997), or the Java Modeling Tools (JMT) (Bertoli et al., 2009). Both tools support both simulative solution as well as analytical solution techniques such as Mean Value Analysis (MVA) (Bolch et al., 1998).

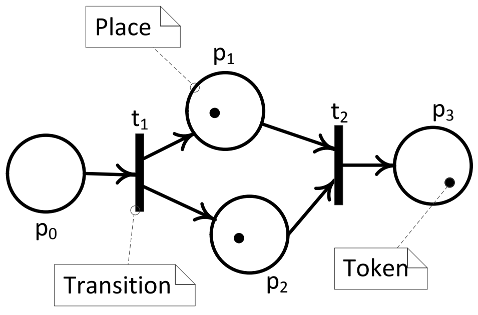
QNs provide a powerful method for modeling contention due to processing resources, i.e., hardware contention and scheduling strategies. For certain classes of QNs, there are efficient analysis methods available. However, QNs are not suitable for representing blocking behavior, synchronization of processes, simultaneous resource possession, or asynchronous processing (Menascé et al., 1994; Kounev, 2005). There are extensions of QNs such as Extended QNs by Bolch et al. (1998) that provide some support to overcome the mentioned drawbacks, but they are considered "rather restrictive and inaccurate" (Kounev, 2005). For more details on QNs, we refer to (Jain, 1991; Bolch et al., 1998; Trivedi, 2002).

Stochastic Petri Nets (SPNs)

Petri Nets (PNs) have been introduced by Petri (1962). An ordinary PN is a bipartite directed graph. The vertices are either places, illustrated as circles, or transitions, illustrated as bars. The (directed) edges connect a place and a transition, and are illustrated as arcs. An initial marking of a PN is a function that describes for each place how many *tokens* the place contains. A transition is enabled to *fire*, if each input place, i.e., a place for which there is a directed connection *from* the place *to* the transition, contains at least one token. The *firing* of the transition then destroys a token in each input place and creates a token in all output places, i.e., places where there is a directed connection *from* the transition *to* the place. Figure 2.2 illustrates an ordinary PN with four places and two transitions, before and after the firing of a transition. Note that we described the default behavior of a firing transition, generally, the number of tokens that are destroyed in the input places and created in the output places is not restricted to one. For each transition, this number configurable.



(a)



(b)

Figure 2.2: Ordinary Petri Net (a) Before Firing and (b) After Firing Transition t_1

In contrast to QNs, PNs are designed to model synchronization behavior of concurrent requests via shared places. However, ordinary PNs do not provide means to model any timing behavior. Hence, several extensions have been proposed to augment PNs with timing aspects (see Kounev (2005)). In particular, SPNs (Bause and Kritzinger, 2002) add an exponentially distributed firing delay to each transition. The delay specifies the time the transition waits after being enabled before it fires. There are also Generalized Stochastic PNs (GSPNs) that introduce two types of transitions, namely immediate transitions and timed transitions. Immediate transitions fire in zero time. If at a point in time more than one immediate transition is enabled, the transition to fire next is chosen based on configured firing weights. Timed transitions fire after an exponentially distributed delay as in SPNs. For formal definitions of the above-mentioned extensions to PNs, we refer to (Bause and Kritzinger, 2002).

While SPNs are a powerful tool to model blocking and synchronization behavior, it is difficult to model waiting lines for processing resources with respect to typical scheduling strategies such as FCFS.

Queueing Petri Nets (QPNs)

Bause (1993) introduces QPNs as a merger of QNs and SPNs. QPNs combine the advantages of QNs and SPNs, and thus eliminate their respective disadvantages. Briefly, the combination works as follows: The places of an SPN can be enriched by queues as they are described as part of a QN. As a result, resources that are processed using certain scheduling strategies as well as simultaneous resource possession, synchronization and blocking can be easily modeled.

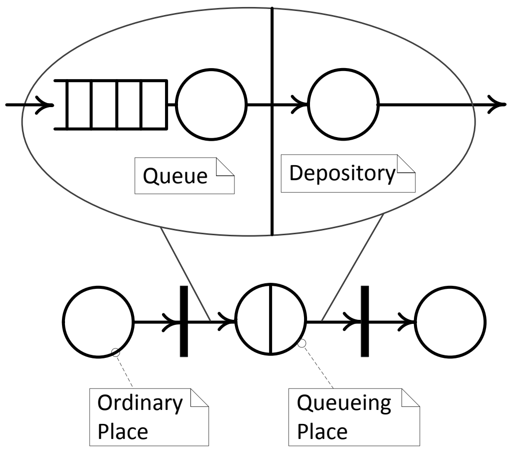


Figure 2.3: Queueing Place as Part of a Queueing Petri Net, cf. Kounev (2005)

Figure 2.3 depicts such a *queueing place.* In contrast to an ordinary place, a queueing place consists of two parts, namely a queue and a depository for tokens that have completed their service at the queue. Tokens entering a queueing place are passed to its queue, tokens leaving a queueing place are taken from its depository. For a formal introduction of QPNs, see Section 5.3.1. In that section, the formalization is needed to formally describe transformations to QPNs.

Available tools for the analytical solution of QPNs are based on Markov chain analysis (Bause and Kritzinger, 2002) and thus can suffer from the state space explosion problem. “QPN models of realistic systems are too large to be analyzable using currently available analysis techniques” (Kounev, 2005). SimQPN is a tool supporting the analysis of QPNs by optimized discrete-event simulation (Kounev and Buchmann, 2006; Spinner et al., 2012). In our work, we thus use SimQPN as QPN model solver.

Layered Queueing Networks (LQNs)

LQNs implement the concept of layered performance models (Franks, 1999). While classical QNs do not allow modeling simultaneous resource possession, LQNs belong to the class of Extended QNs that support that feature. LQNs model layered systems consisting of *tasks* and *processors.* Processors represent physical resources, such as CPUs or disk drives. Tasks are the main interacting entities in LQNs, they can represent software entities, software resources, services of hardware resources, but also load generating users that cycle in an endless loop and create requests for other tasks.

Tasks are arranged in a layered hierarchy. Both processors and tasks have a request queue and a corresponding request scheduling strategy. A task may have one or more *entries* modeling the services it provides. An entry either directly specifies a resource demand to the task's (host) processor or the entry refers to a control flow graph consisting of *activities* that issue resource demands. Entries and activities can also call other entries of tasks that reside in lower layers. Such calls can be synchronous or asynchronous. Resource demands are specified as mean values of exponential distributions. The control flow graphs for activities support sequences, branches, loops, and forks where the branch probabilities and loop iteration numbers are specified as constants. For details on LQNs, we refer to Franks (1999).

LQNs have been applied in many case studies with, e.g., web servers (Dilley et al., 1997), database systems (Sheikh and Woodside, 1997), Enterprise Java systems (Xu et al., 2005), or telecommunication systems (Shousha et al., 1998). As LQN solver tools, there is a simulator named lqsim (Franks et al., 2011) as well as an analytical solver based on approximate MVA named LQNS (Franks et al., 2011; Franks, 1999).

Stochastic Process Algebras (SPAs)

As another formalism for performance modeling, SPAs were first proposed in Herzog (1990). SPAs have explicit operators with formally defined semantics and thus offer a formally defined compositionality which is in contrast to QNs and PNs. They extend classical process algebras, e.g., CCS (Milner, 1989), with stochastic times and probabilistic choice (Clark et al., 2007). For example, PEPA (Hillston, 1996) supports exponential distributions as timing values. We do not go into detail here, since SPAs are not used in the context of this thesis.

Summary

A survey of model-based performance prediction techniques was published in Balsamo et al. (2004). A number of techniques utilizing a range of different performance models have been proposed including (product-form) Queueing Networks (e.g., Menasce and Virgilio (2000)), Extended Queueing Networks (e.g., Cortellessa and Mirandola (2000)), Stochastic Petri Nets (e.g., López-Grao et al. (2004)), Queueing Petri Nets (e.g., Kounev (2006)), Layered Queueing Networks (e.g., Franks (1999)), Stochastic Process Algebras (e.g, Gilmore et al. (2005)) and general simulation models (e.g., Balsamo and Marzolla (2003)). Such models capture the temporal system behavior and can be used for performance prediction by means of analytical or simulation techniques.

However, these predictive performance models are normally used as high-level system performance abstractions and as such they do not explicitly distinguish the degrees-of-freedom and performance-influencing factors of the system's software architecture and execution environment. They are high-level in the sense that: i) complex services are modeled as black boxes without explicitly capturing their internal behavior and the influences of their deployment context, configuration settings and input parameters, and ii) the execution environment is abstracted as a set of logical resources (e.g., CPU, storage, network) without explicitly distinguishing the performance influences of the various layers (e.g., physical infrastructure, virtualization, and middleware) and their configuration.

2.3 Descriptive Architecture-Level Performance Models

In this section, we present performance models that reflect the architectural structure of a software system, referred to as *descriptive architecture-level* performance models. They are detailed models that explicitly capture the software architecture and configuration. Typically, they are software architecture models annotated with descriptions of the system's performance-relevant behavior. In general, architecture-level performance models are built during system development and are used at design and deployment time to evaluate the performance behavior of alternative system designs and/or predict the system performance for capacity planning purposes. For that purpose, the common goal is to enable the automated transformation of such architecture models into prediction models making it possible to predict the system performance and resource consumption for a fixed workload and configuration scenario.

A number of architecture-level performance meta-models have been developed by the performance engineering community in the last years. The most prominent examples are the UML Profile for Schedulability, Performance and Time (UML-SPT) profile (Object Management Group (OMG), 2005) and its successor the UML MARTE profile (Object Management Group (OMG), 2006), both of which are extensions of the Unified Modeling Language as the de facto standard modeling language for software architectures. Other proposed meta-models include, e.g., CBML (Wu and Woodside, 2004), KLAPER (Grassi et al., 2007), PCM (Becker et al., 2009), and ROBOCOP (Bondarev et al., 2004). A recent survey of model-based performance modeling techniques for component-based systems was published in Koziolek (2010).

The approaches differ from each other in the following aspects (Koziolek, 2010):

- The level of detail at which the meta-models support service behavior modeling.
- The flexibility of how model parameters such as resource demands can be specified.
- The capability to describe dependencies between service input parameters and model parameters.
- The support for describing a component's state.

In the following, we discuss these aspects for relevant approaches.

Bertolino and Mirandola (2004) present a performance analysis approach for component-based systems, named Component-Based SPE (CB-SPE), based on the architecture-level abstractions of UML-SPT (Object Management Group (OMG), 2005). In UML 2.0, a software component is modeled as an extended class. Component service behavior can be modeled with standard UML's collaboration, sequence and activity diagrams. Performance-related properties are annotated via the profile extension mechanism of UML. CB-SPE transforms UML models to QN models. Component services are annotated with constant resource demands that are parameterized with respect to platform parameters but independent of service input parameters. Components are composed via sequence diagrams that also determine the service control flow. The control flow is thus modeled component-externally, i.e., outside the component boundaries. The approach does not support components consisting of nested components, and each time a component is replaced with a different component, the modeling steps of the involved sequence diagrams have to be repeated. Given the complexity of UML and its "semi-formal" (Koziolek, 2010) semantics, transformations of UML models to predictive performance models typically consider only limited parts of UML. Furthermore, UML-based approaches are difficult to apply in a component-based development process because they do not naturally support replacing components in the models. Moreover, UML lacks concepts to specify parameterizable models, which are desired for software components.

Wu and Woodside (2004) present an extension of LQNs that allow modeling software components. The approach is called Component-Based Modeling Language (CBML). The concept of a task of an LQN is enriched by a so-called slot so that a task represents a component. A slot contains one or more interfaces describing services the component provides, respectively requires. Components consisting of sub-components are supported. Component-internal control flow can be modeled in a fine-grained fashion with, e.g., branches or loops. Resource demands are specified only as exponential distributions. Dependencies on service input parameters cannot be specified.

Hissam et al. (2001, 2002) and Wallnau and Ivers (2003) present an approach called Prediction Enabled Component Technology (PECT). It is a framework to predict, e.g., service response times for component-based software systems. It is a methodology describing how to augment component technologies with analysis and prediction technologies. In Wallnau and Ivers (2003), the authors use the Component Composition Language (CCL) that allows describing "structural, behavioral and analysis-specific information about component technologies" (Wallnau and Ivers, 2003). The structural description of a component consists of provided and required services. The behavior of a service can be described with a variation of a UML statechart. Both synchronous and asynchronous communication is supported. Moreover, components containing inner components are also supported. However, Wallnau and Ivers (2003) does not mention flexible parameter characterization by, e.g., observed distributions, or dependencies between parameters.

Grassi et al. (2007) present a method for the analysis of non-functional attributes of component-based systems. A language named Kernel LAnguage for PErformance and Reliability analysis (KLAPER) is described that aims to bridge the gap between design-oriented (i.e., architecture-level) notations and analysis-oriented (i.e., predictive) formalisms. KLAPER serves as intermediate model between those two types of notations. Intermediate models cope with the problem of having of n input formats and m output formats (*N-to-M problem*). Instead of defining $n \cdot m$ transformations for each pair of architecture-level notation and predictive formalism, for each architecture-level notation a transformation *to* KLAPER should be defined and for each predictive formalism a transformation *from* KLAPER should be defined. Thus, only $n+m$ transformations need to be specified. For instance, in Grassi et al. (2007), a transformation from UML to KLAPER, as well as a transformation from KLAPER to Extended QNs, is defined. The focus of KLAPER is on performance and reliability prediction. The language does not distinguish between hardware resources and software components. Software components containing inner components cannot be described. Service behavior can be modeled using control flow constructs such as branches, loops and forks. Dependencies between parameters are not considered in KLAPER. Parameters can be characterized with observed distributions.

Bondarev et al. (2004) present an approach to performance prediction based on the ROBOCOP component model. The approach aims to predict real-time properties such as response time, blocking time and number of missed deadlines per task, already in the design-phase of a component-based embedded real-time system. The ROBOCOP component model supports the notion of provided and required interfaces. Components containing nested components are not supported. Service control flow is modeled as a sequence of synchronous or asynchronous tasks. The effect of input parameters on service behavior and resource usage is modeled explicitly, however, stochastic parameter characterizations are not supported. Resource demands are described as constants. Moreover, the resource model is very simple and is not suitable for distributed systems.

Sitaraman et al. (2001) propose a formal approach to enable compositional performance predictions for component-based software systems. It is based on a variation of the RESOLVE (Sitaraman and Weide, 1994) specification language for software components. The authors aim at *verifying* the performance behavior of a component against a given specification. A component is specified by a list of service signatures together with pairs of pre- and postconditions. Required interfaces of a component are not made explicit. Service control flow is not modeled. Service execution times and memory consumptions of a component service are specified in extended O-notation, parameterized by service input parameters. Further dependencies between parameters are not considered. Moreover, the resource model is implicit and fixed, e.g., it does not allow distinguishing different resource types. Blocking behavior due to semaphores is also not considered.

Hamlet (2009) presents a formal testing-based theory to predict functional and non-functional properties (performance and reliability) of a component-based software system. The underlying component model imposes several restrictions on the notion of a component in order to reduce the complexity of the compositional theory. A software component is a mathematical function with input from floating point value domains. Components may have local persistent state, represented by a single floating point value. The input domain of a component is partitioned in subdomains. For each subdomain, the execution time is determined. Furthermore, the call propagation needs to be determined, i.e., which output maps to which input subdomains of connected components. Based on this information, the execution time of the component-based system can be predicted. The theory does not consider concurrency or blocking behavior. The resource model is implicit due to the obtained execution times. Dependencies between parameters as well as stochastic parameter characterizations are not supported.

Mos and Murphy (2002a,b) present a monitoring approach to derive performance predictions. It is a framework called COMPAS for the performance management of Java Enterprise Edition (Java EE) applications. Performance data is extracted from a running application and used to generate interaction models that describe the system behavior. An explicit context model is not defined. In this approach, components are abstractions of Enterprise JavaBeans (EJBs). The components are represented by UML models with UML-SPT annotations. Service control flow is modeled according to the control flow that is visible at the component boundaries, i.e., component-internals are not considered. The implementation of COMPAS assumes synchronous invocation style, i.e., it considers session beans and entity beans but no message-driven beans (Sriganesh et al., 2006). In order to reduce the total overhead of monitoring, an adaptive monitoring concept is proposed that keeps the amount of monitoring small while detecting performance anomalies. Parameter dependencies are not considered.

Diaconescu and Murphy (2005) describe an extension of the COMPAS framework. The authors develop an Automatic Quality Assurance (AQuA) framework that aims at adapting component-based applications at run-time if a performance issue is observed. As adaptation action, the replacement of a component with a functionally equivalent component with different performance behavior is considered. AQuA involves the observation of response time requirements. A violation of such a requirement then triggers the adaptation process. The adaptation process decides on the model level which component should serve as replacement of a component that has been identified as bottleneck.

Becker et al. (2006) argue that current component models do not reflect the influence of the deployment context on the component behavior sufficiently. The authors advocate an explicit context model as part of the component specification that captures the dependencies of functional and extra-functional properties on inter-component relationships, the execution environment, the allocated hardware and software resources, as well as the usage profile. A modeling notation based on extensions of UML-SPT is proposed by Koziolek et al. (2006) allowing component developers to explicitly specify the influence of parameters on the component resource demands as well as on their usage of external services. A parametric contract in the form of a so-called service effect specification is specified for each component service describing its behavior and control flow in an abstract and parametric manner (Reussner, 2001).

The above approaches were combined in the Palladio Component Model (PCM), a domain-specific modeling language for describing performance-relevant aspects of component-based software architectures (Becker et al., 2009). It provides modeling constructs to capture the influence of the following four factors on the system performance: i) the implementation of software components, ii) the external services used by components, iii) the component execution environment, and iv) the system usage profile. These performance-influencing factors are reflected in the above-mentioned explicit context model and parameterized component specifications. Recent surveys (Becker et al., 2004; Koziolek, 2010) show that the clear separation of these factors is one of the key benefits of PCM compared to other architecture-level performance models such as the UML-SPT and MARTE profiles (Object Management Group (OMG), 2006), CB-SPE (Bertolino and Mirandola, 2004), or KLAPER (Grassi et al., 2007). PCM models are divided into five sub-models: The *repository model* consists of interface and component specifications. A component specification defines which interfaces the component provides and requires. For each provided service, the component specification contains a high-level description of the service's internal behavior. The description is provided in the form of a so-called Resource Demanding Service Effect Specification (RDSEFF). The *system model* describes how component instances from the repository are assembled to build a specific system. The *resource environment model* specifies the execution environment in which the system is deployed. The *allocation model* describes the mapping of components from the system model to resources defined in the resource environment model. The *usage model* describes the user behavior by capturing the services that are called, the frequency (workload intensity) and order in which they are invoked, and the input parameters passed to them.

Techniques to solve PCM models include the SimuCom simulator (Becker et al., 2009) that implements a process-based discrete-event simulation. Technically, it is based on a model-to-text transformation that maps PCM instances to Java code. The code is then loaded by SimuCom and executed during a simulation run. As part of the SLAstic adaptation framework (van Hoorn et al., 2009), there is another simulator for PCM instances, named SLAstic.SIM (von Massow et al., 2011). In contrast to SimuCom, it implements an open workload simulation that is driven by workload traces that may have been generated or recorded prior to the simulation. In addition, the simulator supports reconfiguration operations during simulation, e.g., it supports the migration and (de-)replication of software components as well as the (de-)allocation of resource containers. In SLAstic, these capabilities are used to plan and execute system adaptations on the model-level in order to support online capacity management of component-based software systems (van Hoorn, 2014b). Given that it is based on PCM, the framework is not focused on performance prediction at run-time. Another performance engineering approach based on PCM is SimuLizar(Becker et al., 2013a). SimuLizar is a design-time modeling and prediction approach for self-adaptive systems (Becker et al., 2013a). It aims at enabling performance prediction of self-adaptive systems over their various configurations, allowing the analysis of also the transient adaptation phases. Furthermore, SimuLizar enables a semi-automatic analysis of whether non-functional requirements, formally specified using RELAX (Whittle et al., 2009), are met or not. It can analyze the gradual fulfillment of requirements during the simulation of the modeled system (Becker et al., 2013b). That way, it provides feedback for software engineers at design-time, i.e., not at run-time, to iteratively improve the design of self-adaptive systems.

Summary

Approaches that explicitly consider the influence of parameters in performance analysis are, e.g., Becker et al. (2009); Bondarev et al. (2004); Wu and Woodside (2004); Bertolino and Mirandola (2004). The approaches in Bertolino and Mirandola (2004); Wu and Woodside (2004) lack expressiveness for capturing dependencies between parameters. For example, they do not consider service input and output parameters, or limit the set of parameters to, e.g., thread pool size or resource demand parametrization. The most advanced approaches concerning parameter dependencies are Becker et al. (2009); Bondarev et al. (2004). Components and their behavior can be specified in a parameterized way, considering the influence of input and deployment specific parameters on the component's resource demand, control flow, and so on. Orthogonal approaches tackling the challenge of parametric dependencies in performance analysis are Hamlet (2009) and Hissam et al. (2002). In contrast to the other approaches, they model the component's internal state, however, they do not differentiate the execution time on different resources (CPU, HDD) or omit the specification of required interfaces.

While PCM (Becker et al., 2009) provides a good basis to model the performance behavior of a component in a parameterizable and compositional manner, it shares the same restriction as the other architecture-level modeling approaches mentioned above: It is used at design-time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes. However, as described in the next section, differences between offline and online scenarios for performance prediction lead to different requirements on the underlying performance abstractions of the system architecture suitable for use at design-time versus at run-time.

2.4 Run-Time versus Design-Time Performance Prediction

There are some fundamental differences between offline and online scenarios for performance prediction. This leads to different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques. In the following, we summarize the main differences in terms of goals and underlying assumptions, driving the evolution of design-time models and run-time models, respectively.

Evaluating Design Alternatives versus Evaluating Impact of Changes

At system design-time, the main goal of performance modeling and prediction is to evaluate and compare different design alternatives in terms of their performance properties.

In contrast, at run-time, the system design (i.e., architecture) is relatively stable and the main goal of online performance prediction is to predict the impact of dynamic changes in the environment (e.g., changing workloads, resource allocations, service compositions).

Aligning the Model Structure to Developer Roles versus System Layers

Given the goal to evaluate and compare different design alternatives, design-time models are typically structured around the various developer roles involved in the software development process (e.g., component developer, system architect, system deployer, domain expert), i.e., a separate sub-meta-model is defined for each role. In line with the component-based software engineering paradigm, the assumption is that each developer with a given role can work independently from other developers and does not have to understand the details of sub-meta-models that are outside of their domain, i.e., there is a clear separation of concerns. Sub-meta-models are parameterized with explicitly defined interfaces to capture their context dependencies. Performance prediction is performed by composing the various submodels involved in a given system design. To summarize, at design-time, model composition and parameterization is aligned with the software development processes and developer roles.

At run-time, an implemented and deployed system is already available and a strict separation and encapsulation of concerns according to the developer roles is no longer that relevant. Instead, given the dynamics of modern systems, it is more relevant to be able to distinguish between static and dynamic parts of the models. The software architecture is usually stable, however, the system configuration (e.g., deployment, resource allocations) at the various layers of the execution environment (virtualization, middleware) may change frequently during operation. Thus, in this setting, it is more important to explicitly distinguish between the system layers and their dynamic deployment and configuration aspects, as opposed to distinguishing between the developer roles. Given that performance prediction is typically done to predict the impact of dynamic system adaptation, models should be structured around the system layers and parameterized according to their dynamic adaptation aspects.

Type and Amount of Data Available for Model Parameterization and Calibration/Adjustment

Performance models typically have multiple parameters such as workload profile parameters (workload mix and workload intensity), resource demands, branching probabilities, and loop iteration frequencies. The type and amount of data available as a basis for model parameterization and calibration/adjustment at design-time versus run-time is substantially different.

At design-time, model parameters are often estimated based on analytical models or measurements if implementations of the system components exist. On the one hand, there is more flexibility since in a controlled testing environment, one could conduct arbitrary experiments under different settings to evaluate parameter dependencies. On the other hand, possibilities for experimentation are limited since often not all system components are implemented yet, or some of them might only be available as a prototype. Moreover, even if stable implementations exist, measurements are conducted in a testing environment that is usually much smaller and may differ significantly from the target production environment. Thus, while at design-time, one has complete flexibility to run experiments, parameter estimation is limited by the unavailability of a realistic production-like testing environment and the typical lack of complete implementations of all system components.

At run-time, all system components are implemented and deployed in the target production environment. This makes it possible to obtain much more accurate estimates of the various model parameters taking into account the real execution environment. Moreover, model parameters can be continuously calibrated to iteratively refine their accuracy. Furthermore, performance-relevant information can be monitored and described at the component instance level and not only at the type level as typical for design-time models. However, during operation, we don't have the possibility to run arbitrary experiments since the system is in production and is used by real customers placing requests. In such a setting, monitoring has to be handled with care, keeping the monitoring overhead within limits (non-intrusive approach) such that system operation is not disturbed. Thus, at run-time, while theoretically much more accurate estimates of model parameters can be obtained, one has less control over the system to run experiments and monitoring must be performed with care in a non-intrusive manner.

Trade-Off Between Prediction Accuracy and Overhead

Normally, the same model can be analyzed (solved) using multiple alternative techniques such as analytical techniques providing exact results, numerical approximation techniques, simulation and bounds analysis techniques. Different techniques offer different trade-offs between the accuracy of the provided results and the overhead of the analysis in terms of elapsed time and computational resources.

At design-time, there is normally plenty of time to analyze (solve) the model. Therefore, one can afford to run detailed time-intensive simulations to obtain accurate results with certain confidence levels.

At run-time, depending on the scenario, the model may have to be solved within seconds, minutes, hours, or days. Therefore, flexibility in trading-off between accuracy and overhead is crucial. The same model is typically used in multiple different scenarios with different requirements for prediction accuracy and analysis overhead. Thus, run-time models must be designed to support multiple abstraction levels and different analysis techniques to provide maximum flexibility at run-time.

Degrees-of-Freedom

The degrees-of-freedom when considering multiple design alternatives at system design-time are much different from the degrees-of-freedom when considering dynamic system changes at run-time such as changing workloads or resource allocations.

At design-time, one virtually has infinite time to vary the system architecture and consider different designs and configurations. At run-time, the time available for optimization is normally limited and the concrete scenarios considered are driven by the possible dynamic changes and available reconfiguration options. Whereas the system designer is free to design an architecture that suits his requirements, at run-time the boundaries within which the system can be reconfigured are much stricter. For example, the software architecture defines the extent to which the software components can be reconfigured and the hardware environment may limit the deployment possibilities for virtual machines or services. Thus, in addition to the performance influencing factors, run-time models should also capture the available system reconfiguration options and adaptations strategies.

2.5 Monitoring Tools

In this section, we provide an overview of monitoring approaches that are currently used in industry and academia to collect performance-relevant measurements at run-time. This overview is relevant when it comes to the topics of model extraction and model maintenance in Chapter 6.

A number of tools exist that aid in monitoring and correlating the performance of software services with the consumption of software resources. Paradyn (Miller et al., 1995) is a performance measurement tool for parallel and distributed programs that collects the data through on-the-fly instrumentation of the application and kernel using POSIX[1] thread utilities. Tuning and Analysis Utilities (TAU) (Shende and Malony, 2006) gathers performance information through code instrumentation of functions, methods, basic blocks, and statements. The collected data can then be analyzed using the tool PerfExplorer2, which provides visualizations, rule sets to interpret performance results, and other capabilities. The Paraver toolkit (Barcelona Supercomputing Center, 2014) organizes the performance data to focus the optimization effort. This tool can be used to visualize and analyze event trace files coming from a variety of sources, focusing on various parallel environments. NetLogger (Gunter and Tierney, 2003) is a set of tools for performance analysis of distributed systems. It includes a library to aid in log generation, as well as visualization and analysis tools, including a lifeline visualization that tracks a data object through time. Host level monitoring in a virtualized environment has not been widely addressed in the literature, and most of the tools focus on Xen technology. For instance, Xenmon (Gupta et al., 2005) reports the resource usage of different Virtual Machines (VMs) and provides an insight into shared resource access and resource scheduling in Xen.

Understanding the performance behavior of distributed applications and the intrinsic relationships between them is fundamental to performance management. Anderson et al.

[1]IEEE Std 1003.1-1988

(2009) presents Chirp, a collection of techniques that tackle three of the main challenges of tracing in distributed systems by including: i) a low-overhead tracing mechanism capable of operating in large systems without impacting their behavior or performance, ii) a technique for producing highly accurate time synchronization across hosts, surpassing the Network Time Protocol (NTP) accuracy, and iii) scalable data processing techniques that assist in the analysis of large traces in the order of billions of trace points. A highly cited work that reports the development of an automatic tool that extracts system workload characteristics is Magpie (Barham et al., 2004). Using low-overhead instrumentation, it first records fine-grained events generated by kernel, middleware, and applications. Then, it correlates these events to the control flow and to the resource consumption of the requests. However, in contrast to Chirp, Magpie does not address cross-machine time calibration. Profiling has proven to be very helpful for performance debugging of stand-alone programs. Call graph profiling (Graham et al., 1982) and call path profiling (Hall, 1992) are two widely used approaches to profiling stand-alone programs. Graham et al. (1982) introduces gprof, a profiler that provides caller-context information, that is, the set of conditions and facts that enclose an operation call. Software operation executions are embedded in sequences of interacting operation executions that participate in replying to external service requests. There are three models of the general calling-context approach that take into account different aspects of the execution, namely caller-context, stack-context, and trace-context. Most modern profiling tools, such as Intel's VTune Performance Analyzer follow the approach of gprof by providing caller-context information (Intel, 2013). The trace-context analysis used in Kieker (Rohr et al., 2008) is an extension of the concept of caller-contexts.

In complex distributed applications, it is important for the monitoring to collect enough information for measuring business transactions not just as experienced by the end user, but in enough detail to identify sub-transactions executed by the different application components, their dependencies, and their contribution to the overall response time. Controlling the monitoring overhead is of obvious importance to performance monitoring tools. Overhead may result in non-responsive systems and failure to adhere to Service Level Agreements (SLAs). There are several approaches to address this, from providing the user with tools to control the monitoring level before or during the monitoring (Mos and Murphy, 2002a), to cost systems that report the perturbation introduced by the instrumentation or control the level of instrumentation according to a predefined maximum cost (Hollingsworth and Miller, 1996). A more subtle problem is that even relatively low overhead may have disproportionate effect on important performance metrics, having an impact on the performance analysis (Mytkowicz et al., 2007). In addition to overhead reduction, overhead compensation may alleviate those issues (Malony et al., 2007).

2.6 Summary

In this chapter, we presented different existing approaches to performance prediction. They differ in the employed abstraction levels, the phases of the system lifecycle they are targeting, and the input parameter space they consider when conducting performance predictions.

We started with black-box approaches that abstract the system of interest at a very high level, i.e., without taking any details about the internal system structure into account. We then presented classical predictive performance models such as QNs or QPNs. Such models capture the temporal system behavior and can be used for performance prediction by means of analytical or simulation techniques. However, they are normally used as high-level system performance abstractions and as such they do not explicitly distinguish the degrees-of-freedom and performance-influencing factors of the system's software architecture and execution environment. This is in contrast to the descriptive architecture-level performance models we described in Section 2.3. Such models are software architecture models annotated with descriptions of the system's performance-relevant behavior. Architecture-level performance models are typically built during system development and are used at design and deployment time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes.

Architecture-level performance models provide a powerful tool for performance prediction, but they are typically developed for use at design-time. We argued that there are fundamental differences between offline and online scenarios for performance prediction. This leads to different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques suitable for use at design-time versus run-time. In Section 2.4, we summarized the main differences in terms of goals and underlying assumptions driving the evolution of online models.

Finally, in Section 2.5, we provided an overview of current monitoring technologies and tools that are used in industry and academia to collect performance-relevant measurements at run-time. This is related to the efforts we present in Chapter 6 where we describe methods to integrate architecture-level performance models and system environments with the goal to keep models up-to-date during operation as the system evolves.

3. Related Work

The online performance prediction approach presented in this thesis is based on architecture-level performance models. Thus, it is related to: (i) the research area of architecture-level performance models as well as to (ii) the research area of online performance prediction. Note that (i) has already been presented and discussed as part of the foundations in Section 2.3 and Section 2.4, respectively. In this chapter, Section 3.1 focuses on (ii), i.e., approaches targeted at online performance prediction. Section 3.2 discusses related work in the area of performance model extraction and maintenance.

Figure 3.1 illustrates the described structure. As depicted on the x-axis of the diagram, approaches to performance prediction can be distinguished by the underlying modeling approach, i.e., if the prediction approach uses black-box models, predictive stochastic performance models, or descriptive architecture-level performance models. The y-axis of the diagram depicts the application domains of design- and deployment-time performance prediction as well as online performance prediction. While Chapter 2 focused on describing the different types of performance models, in this chapter, the emphasis is on the domain of online performance prediction.

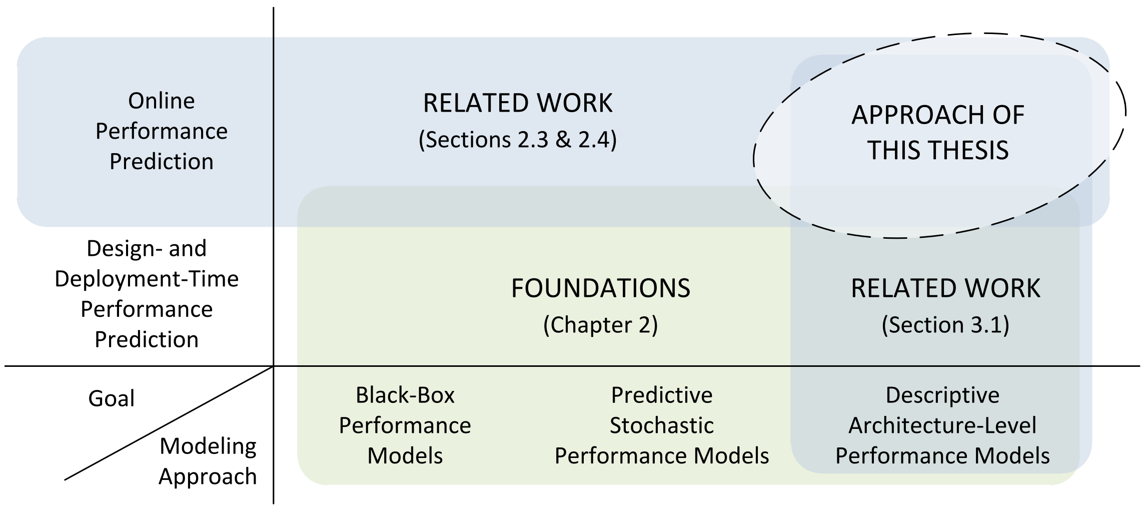

Online Performance Prediction
RELATED WORK (Sections 2.3 & 2.4)
APPROACH OF THIS THESIS
Design- and Deployment-Time Performance Prediction
FOUNDATIONS (Chapter 2)
RELATED WORK (Section 3.1)
Goal
Modeling Approach
Black-Box Performance Models
Predictive Stochastic Performance Models
Descriptive Architecture-Level Performance Models

Figure 3.1: Related Work and Foundations

Besides related approaches concerned with performance prediction mechanisms, this chapter also presents approaches that are related to the work we present in Chapter 6: The integration of performance models with the system environment relates to the area of performance model extraction and maintenance. This is discussed in Section 3.2.

3.1 Online Performance Prediction

Many approaches to online performance prediction have been developed in the research community over the past decade. They are mostly published as part of an online capacity management approach. Such approaches are typically based on either black-box models, such as control theory feedback loops and machine learning techniques (see Section 2.1 for details), or predictive stochastic performance models such as Queueing Networks (QNs), Layered Queueing Networks (LQNs), and Queueing Petri Nets (QPNs) (see Section 2.2). In the following, we present some of the most prominent approaches and describe which degrees-of-freedom they support for performance prediction, i.e., if the impact of service re-compositions, changes of resource allocations, usage profile changes, or load-intensity changes can be predicted, as formulated in the problem statement in Section 1.2.

3.1.1 Approaches Using Black-Box Models

Examples for approaches based on feedback loops and control theory are, e.g., Abdelzaher et al. (2002); Almeida et al. (2010). Abdelzaher et al. (2002) uses classical feedback control theory to achieve overload protection and satisfy service response time objectives. Response times and throughputs can be predicted for load intensity changes. The same applies for Almeida et al. (2010) where the technique is used to address resource allocation and admission control problems in virtualized servers. Furthermore, Almeida et al. (2010) claim that an approach using a control theory based feedbacks loop can normally guarantee system stability by capturing the transient system behavior.

Examples of approaches based on machine learning techniques are, e.g., Tesauro et al. (2006); Kephart et al. (2007); Elkhodary et al. (2010); Gambi et al. (2013); Shivam et al. (2006); Mi et al. (2010); Jamshidi et al. (2014). Machine learning techniques may capture the system behavior based on observations at run-time without the need for an a priori analytical model of the system.

Tesauro et al. (2006) uses a knowledge-free trial-and-error methodology (reinforcement learning) to learn resource valuation estimates and construct policies for allocation decisions. In fact, it is a hybrid approach since both reinforcement learning and queueing models are applied. Offline training is used, before the reinforcement learning module is trained online, based on the consequences of its own decisions. The offline training data is collected while a queueing model policy makes management decisions in the system. The approach only supports performance-aware scheduling of http requests to web servers. Further allocation or configuration changes are not considered.

Kephart et al. (2007) uses reinforcement learning to learn black-box models that manage the trade-off between performance and power consumption. Like in Tesauro et al. (2006), the approach aims at providing a scheduling mechanism that distributes http requests to web servers. Service response time objectives should be met under energy efficiency considerations. The impact of, e.g., allocation or configuration changes cannot be analyzed using the underlying models.

In Elkhodary et al. (2010), a reinforcement learning-based approach is used to implement a self-adaptive software system. Instead of using an explicit model, the approach is intended to learn the impact of adaptation decisions concerning the system's operational goals such that this knowledge can be used for future adaptation decisions. In the provided case study, the approach learns dependencies where the independent variables are, e.g., request arrival rate, authentication mechanism (per session vs. per request authentication), and caching (on vs. off), and the dependent variable is the request's response time. However, resource allocations and resource utilization predictions are not considered.

Gambi et al. (2013) applies Kriging models to predict system behavior within a control loop. The independent variables are workload and system configuration data, the dependent variables are application-level performance metrics such as average response times. The decision maker module of the control loop queries the Kriging models to predict the anticipated performance evolution for a given workload and system configuration. By re-querying the Kriging models, a target configuration is searched. In the case study, the workload and system configuration data consists of the number of incoming and queued jobs and the number of application server nodes. The dependent variable is the average request response time. Fine-grained service reconfigurations or resource allocations are not considered.

An active learning approach to resource allocation for simple batch workloads is proposed in Shivam et al. (2006). This approach uses performance histories to build black-box models/functions of frequently used applications. The approach, however, is focused on compute batch tasks that run to completion without interruption. Request arrivals and concurrency related behavior are not considered.

Mi et al. (2010) uses a genetic algorithm to reconfigure Virtual Machines (VMs) of a virtualized data center while taking performance guarantees and resource efficiency into account. A chromosome in the genetic algorithm represents a reconfiguration candidate. The fitness functions respect both resource utilization and power consumption. Service response times are not considered. Instead, performance guarantees are defined using CPU resource utilization caps.

Jamshidi et al. (2014) describes a resource provisioning approach that aims at providing and releasing resources in an elastic fashion. The approach exploits fuzzy logic to enable qualitative specification of elasticity rules in order to provide auto-scaling, i.e., adding and removing resource on-the-fly as they are needed. It can be used without offline training because it takes advantage of online incremental learning. An explicit performance model is not derived. In the experiments in Jamshidi et al. (2014), it is shown how the elasticity controller handles unexpected spikes in the workload. However, the approach assumes all involved computing nodes to be stateless.

In Balsamo et al. (2006), a different approach for the performance prediction of component-based software systems is proposed. Asymptotic bounds for system throughput and response time are derived from the software specification without deriving explicit performance models. However, only coarse-grained bounds can be calculated. For instance, concerning service response times in an open workload scenario, only lower bounds are provided.

Furthermore, numerous approaches to resource allocation and performance management in service-oriented Grid computing environments are available in the literature (Foster and Kesselman, 2003; Ali et al., 2004; Othman et al., 2003). These approaches, however, are mostly targeted at scientific computing and are not suitable for enterprise workloads. Simple ad hoc procedures are used to map service requirements to resource requirements. As such these mechanisms do not possess any sophisticated performance prediction capabilities that are required to enforce SLAs.

3.1.2 Approaches Using Predictive Stochastic Performance Models

Predictive stochastic performance models are typically used in the context of utility-based optimization techniques for resource management. They are embedded within optimization frameworks aiming at optimizing multiple criteria such as different Quality of Service (QoS) metrics.

Mistral (Jung et al., 2010) is a resource management framework with a multi-level resource allocation algorithm considering the following allocation actions: adapt a VM's CPU capacity, add or remove a VM, live-migrate a VM between hosts, and shutdown or restart a host. The approach considers power consumption, performance and transient costs in its reconfiguration algorithm. However, the approach is based on a simple multi-tier application with read-only transactions and a fixed web tier modeled by an LQN (Jung et al., 2008) that is solved using simulation. Analytical solutions are not considered to be viable, further details on the performance modeling approach are missing. The extensive evaluation shows promising results, but it is restricted to four different instances of the RUBiS benchmark and does not consider different types of services.

Verma et al. (2008) aims at a power-aware workload placement controller that distributes applications to heterogeneous virtualized server clusters. When deciding where to place an application, the approach takes performance benefits, power consumption as well as migration costs into account. Details on the underlying performance model are not provided.

With the same goal as the above-mentioned approaches, Chen et al. (2005) uses QNs to implement a performance prediction mechanism. The approach assumes a data center of M identical servers. Each application running in the data center is modeled as a G/G/m queue (in Kendall's notation, see Section 2.2 for an explanation). Response time predictions for the applications are derived using approximative analytical solution techniques from Bolch et al. (1998). The latter are only applicable in steady-state situations. The system's transient dynamics is proposed to be captured in a control theoretic approach using a feedback loop. However, the approach only considers the mapping of applications to identical servers, further fine-grained reconfigurations are not considered.

Bennani and Menascé (2005) use analytically solved QNs to address the problem of deploying application environments in data centers so that performance and resource efficiency goals are satisfied. For online transaction applications, the average response time is computed using the basic law $R = \frac{D}{(1-U)}$ (Bolch et al., 1998) where D is the resource demand and U is the resource utilization. However, this law only applies for M/M/1 queues and is thus not applicable in more complex environments. For batch processing applications, an approximative mean value analysis is proposed as a model solving method. The proposed mean value analysis also has the assumption of exponentially distributed service times and arrival rates and is thus not generalizable.

In Zhang et al. (2007), the authors focus on resource provisioning for multi-tier applications. Each tier is modeled as a queue where the number of servers corresponds to the tier's multiplicity. Each application translates to a workload class. CPU demands are obtained using regression-based approximation. The resulting QN is then solved analytically using classical mean-value analysis.

Urgaonkar et al. (2007) also focuses on predicting performance behavior for multi-tier applications using QN models. The structure of the QN is similar to the QN presented in Zhang et al. (2007). The flow of requests is modeled in more detail so that replication at tiers, load imbalances across replicas, caching effects as well as concurrency limits at each tier are captured. As solving algorithm, Urgaonkar et al. (2007) also uses approximative mean-value analysis to derive average response times.

Abrahao et al. (2006) links a cost model and a performance model to provide an automated Service Level Agreement (SLA)-driven capacity management. In contrast to the above-described approaches, response time objectives are not formulated using average response times but using the tails of the involved response time distributions. The performance behavior is modeled using QNs. Two types of queues are investigated, namely M/M/1 and M/G/1 queues where the former runs First-Come-First-Served (FCFS) as scheduling strategy and the latter runs Processor-Sharing (PS) as scheduling strategy. Three different analytical approximations to predict bounds for response time percentiles are investigated. However, complex application control flows over multiple tiers are not supported.

Li et al. (2009) uses LQNs as performance abstractions of applications deployed in a cloud environment. The application layers are reflected in the layers of the LQN. However, the paper does not provide any details about the model solving approach.

Further approaches using QNs are, e.g., Pacifici et al. (2005) and Menascé et al. (2005). Both approaches focus on simple online performance models that are solved analytically, e.g., QNs consisting of one queue. Underlying component architectures or configurations are not taken into account. Similarly, in Menascé et al. (2007, 2004a), the difference between dedicated and non-dedicated resources, on the one hand, and between service requests with different priorities, on the other hand, is not accounted for.

In Menascé and Bennani (2003); Bennani and Menascé (2004), methods for dynamic monitoring and tuning of e-commerce sites based on online performance models are presented. A performance controller is run periodically and it uses QNs together with combinatorial search techniques to determine the best possible system configuration given its current workload. However, the authors assume a single workload class and the configuration parameters that are considered for tuning are limited to concurrency levels at servers and maximum queue lengths. In Bennani and Menascé (2005), the same approach is applied for resource allocation in data centers.

In Berbner et al. (2006); Song et al. (2005), methods for dynamic performance-aware service selection and composition are presented. They use simple heuristic algorithms based on static information about services. Dynamic service aspects such as usage profiles and execution contexts are not taken into account.

The work of Kounev et al. (2007); Nou et al. (2009) proposes a framework for designing autonomic resource managers that have the capability to predict the performance of the Grid components they manage and allocate resources in such a way that SLAs are respected. Dynamically composed QPNs are used to predict the Grid performance for a given resource allocation and load-balancing strategy. However, the employed performance prediction mechanism has limited scalability and is not applicable to realistically-sized environments because of the overhead incurred in generating and analyzing the models. No mechanism currently exists for managing dynamic performance-relevant information about the multi-layered and usually heterogeneous server environment.

3.1.3 Discussion

In summary, current approaches to performance and resource management at run-time suffer from several limitations. In general, the methods used for managing service performance and resource reservations are based on simple models that cannot adequately capture dynamic aspects of the services and their resource environment. The inability of these methods to predict the end-to-end performance of a dynamically composed service is one of the reasons why service composition is rarely done in an automated manner today.

Existing work in the area of online performance prediction mainly uses either models where the system is abstracted as black box or predictive performance models that capture the temporal system behavior but typically neglect the software architecture and configuration. Thus, the mentioned approaches do not explicitly distinguish the degrees-of-freedom and performance-influencing factors of the system's software architecture and execution environment. Furthermore, they often do not capture important parameter dependencies or blocking behavior due to software bottlenecks such as thread pools. Moreover, the performance analyses often impose restrictive assumptions such as a single workload class, single-threaded components, or homogeneous servers. Service demands and request inter-arrival times are often limited to exponential distributions.

3.2 Performance Model Extraction and Maintenance

In Chapter 6, we present methods to integrate architecture-level performance models and system environments with the goal to keep the models and the system synchronized during operation. A semi-automatic extraction of architecture-level performance models based on system request tracing, model structure maintenance in the context of an autonomic resource management process, as well as the derivation of model parameter values, are described in Chapter 6. Moreover, it is discussed how architecture-level performance models can be calibrated and adjusted in order to increase their accuracy.

Such topics relate to the area of performance model extraction and maintenance. In the following, we discuss relevant approaches.

3.2.1 Model Extraction Using Monitoring Data

This section gives a brief overview of existing approaches that monitor and process data that can be used as input for performance model extraction. It considers model extraction approaches from academia and industry.

Tools concerned with the automated identification of hardware and software resources in a system environment are already available in industry. For instance, Hyperic (Hyperic, 2014) or Zenoss (Zenoss, 2014) provide functionalities for the automated discovery of system, network and software properties both inside and outside of virtual machines. However, such tools do not provide monitoring data at the application level. Dynatrace Diagnostics (Rometsch and Sauer, 2008) is an industrial tool for performance and resource management at the application level. It traces transactions of applications deployed in distributed heterogeneous .NET and Java environments. Besides providing a call tree, it also monitors method arguments and provides information about the system's resource utilization. However, an explicit architecture model including software components and their behavior is not considered. Further industrial tools such as IBM Tivoli, CA Wiley Introscope or AppDynamics share these restrictions.

Tools for automatic and adaptive monitoring at the application level proposed in the research community include for example Carrera et al. (2003); Mos and Murphy (2002a); Rohr et al. (2008). They are typically focused on monitoring Java applications, however, they do not provide means to generate performance abstractions. In Carrera et al. (2003), an automatic monitoring framework covering the operating system, Java Virtual Machine (JVM), middleware and application level is presented. After initially defining performance objectives, it automatically traces the execution and collects performance data that enables hotspot and bottleneck detection. In Mos and Murphy (2002b), an adaptive monitoring and performance management framework called COMPAS is presented. This tool is capable of extracting data from a running Enterprise Java application and generating system behavior models. It addresses performance issues related to the Enterprise JavaBean (EJB) layer in Enterprise Java applications. However, generating performance abstractions of application components or context information of scomponents is not considered.

Approaches directly targeting performance model extraction (from black-box models to architecture-level models) are discussed in the following.

Approaches such as Westermann et al. (2012); Courtois and Woodside (2000); Zhang et al. (2013) use systematic measurements to build black-box mathematical models that employ, e.g., genetic optimization techniques. However, these approaches are purely measurement-based, the models serve as interpolation of the measurements, and neither a representation of the system architecture nor its performance-relevant factors and dependencies are extracted.

Approaches to automatically construct performance models, such as queueing networks, at run-time have been proposed in, e.g., Menascé et al. (2007, 2005); Mos (2004). However, these models are rather limited since they abstract the system at a very high level without taking into account its architecture and configuration. Moreover, restrictive assumptions such as a single workload class or homogeneous servers are often imposed. Further predictive performance models are extracted for example in Kounev et al. (2011), where run-time monitoring data is used to generate QPN models. However, the model structure is fixed and preset, and only model parameters are obtained. Extraction of structural information is considered in Briand et al. (2006); Hrischuk et al. (1999); Israr et al. (2007) where LQNs are used as the target performance model. In Briand et al. (2006), UML sequence diagrams are extracted from trace data which is obtained by aspect-oriented instrumentation. In Hrischuk et al. (1999), special traces, called *angio traces*, are recorded. Each distributed operation gets assigned with a unique *angio dye id*. For message ordering, trace event timestamps are used. Once such traces are recorded, a graph representing the message flow is generated. Using a rule-based graph analysis approach the graph is then transformed into an LQN model whereby different interaction types such as synchronous and asynchronous messaging are detected and represented accordingly. Since Israr et al. claim that such "traces are difficult to obtain in practice" (Israr et al., 2007), an approach based on a less restricted trace data format is proposed. It "uses conventional trace data which is available from many tracing tools" (Israr et al., 2007) and does not rely on angio traces containing an angio dye id that is propagated through the system. Instead, so-called *eventInfo* properties of message traces are used to achieve message correlation. Here, eventInfo is not required to uniquely identify messages. However, it should provide information that, together with observed timestamps, makes a robust message correlation possible. Israr et al. (2007) generates LQN models considering different interaction types using an "algorithm which scales up linearly for very large traces" (Israr et al., 2007), in contrast to the algorithm presented in Hrischuk et al. (1999). Pattern matching on trace data is used to differentiate between asynchronous, blocking synchronous, and forwarding communication. Performance model parameters such as resource demands are considered, however, the system architecture and its layers, as well as parameter dependencies, are not modeled explicitly.

In Chouambe et al. (2008), the authors present "ArchiRec", a tool to extract components and their interface boundaries based on static code analysis. Components and interfaces are identified using heuristics that are based on code coupling metrics and a subsequent hierarchical clustering. A related tool Java2PCM (Kappler et al., 2008) extracts component-level control flow from Java code and generates instances of the Palladio Component Model (PCM). Krogmann et al. (2010) proposes an approach to reverse engineer PCM instances for Java applications using both static analysis and dynamic analysis. The component architecture is obtained via ArchiRec and control flow abstractions are obtained by applying machine learning algorithms on run-time monitoring data (Krogmann et al., 2008). By monitoring service call frequencies and parameter values at the interface level, the black-box property of components is preserved. The monitoring data then serves as input for genetic programming that aims at recovering intra-method control flow and explicit parameter dependencies. With regard to obtaining resource demands, ByCounter (Kuperberg et al., 2008b) and microbenchmarks are used to abstract from concrete timing values (Kuperberg et al., 2008a). ByCounter instruments the application for "dynamic counting of executed Java bytecode instructions" (Kuperberg et al., 2008b) and method invocations. Combined with benchmarking target execution platforms at the bytecode instruction level, cross-platform performance predictions are possible. The behavior models are extracted via static and dynamic analysis, however, in a controlled environment and in an offline setting requiring manual instrumentation to extract the application control flow and data flow.

3.2.2 Model Maintenance Using Monitoring Data

In this section, we provide an overview of approaches that maintain online performance models and performance model parameters using monitoring data. Some of the approaches are focused on the topic of resource demand estimation.

An approach that integrates performance models with the system environment to enable dynamic resource allocation is presented, e.g., in Adam and Stadler (2006). The authors describe a decentralized architecture of a performance-aware middleware that dynamically partitions resources within a large-scale cluster with the goal to optimize a global utility function. The proposed architecture, however, assumes identical servers and a single request class, the maintained model is thus very simplistic.

In Kounev et al. (2007); Nou et al. (2009), see also Section 3.1.2, the QPN performance models that are part of the proposed autonomic resource managers are maintained by continuously re-estimating their resource demands using response time approximations. However, structural changes due to complex adaptation actions are not considered.

The estimation of resource demands to determine representative values can be a challenging and time consuming process given that normally they are not directly measurable and have to be estimated from the available monitoring data (Lazowska et al., 1984). Many approaches to resource demand estimation have been proposed over the years based on different mathematical methods to infer resource demands from measurements on the running system. They apply basic queueing theory laws (such as the Service Demand Law (Menascé et al., 1994; Lazowska et al., 1984), regression techniques (Rolia and Vetland, 1995; Pacifici et al., 2008; Casale et al., 2007; Zhang et al., 2007), and stochastic filtering (Zheng et al., 2008; Kumar et al., 2009a)), or formulate general optimization problems (Menascé, 2008; Zhang et al., 2002; Kumar et al., 2009b). Approaches in this area are investigated and classified as part of Section 6.4.3.

In Menascé et al. (1994), a conventional model calibration technique is introduced. It is conventional in the sense that it is based on comparing the performance metrics (e.g., response time, throughput, and resource utilization) predicted by a performance model against measurements collected in a controlled experimental environment varying the system workload and configuration. Given the lack of control during operation over the system workload and configuration, techniques of this type are not applicable for *online* model calibration. In Liu et al. (2005), performance models are calibrated by application-independent synthetic benchmarks. The approach uses middleware benchmarking to extract performance profiles of the underlying component-based middleware. However, application-specific behavior is not represented explicitly.

The applicability of tracking filters for continuously maintaining model parameters is investigated in Zheng et al. (2005, 2008). They conduct experiments on time-varying systems to assess the ability of the Kalman filter to dynamically adapt its estimates to deterministic and random changes in resource demands. The experiments consider only a single workload class. They conclude that the Kalman filter can adequately track changes in resource demands (Zheng et al., 2005). Furthermore, they ran a set of experiments to determine the influence of the initialization of the process noise covariance matrix and the measurement covariance matrix, as well as the measurement interval length. It is concluded that these parameters can be derived from common system monitoring data with reasonable efforts (Zheng et al., 2005). Kumar et al. (2009a) extends the evaluation of Kalman filters to cases with multiple workload classes. The authors come to the conclusion that the Kalman filter has convergence problems with multiple workload classes because of an underdetermined equation system (Kumar et al., 2009a).

3.3 Summary

In this chapter, we described and discussed approaches that are related to the work presented in this thesis.

Approaches to online performance prediction are discussed in Section 3.1. Current approaches suffer from several limitations. Existing work in this area mainly uses either models where the system is abstracted as black box or predictive performance models that capture the temporal system behavior but typically neglect the software architecture and configuration. Thus, the mentioned approaches do not explicitly distinguish the degrees-of-freedom and performance-influencing factors of the system's software architecture and execution environment. Moreover, the performance analyses often impose restrictive assumptions such as a single workload class, single-threaded components, or homogeneous servers. Service demands and request inter-arrival times are often limited to exponential distributions.

Approaches for the extraction and maintenance of performance models using monitoring data are discussed in Section 3.2. Most approaches extract only simple black-box or simplistic predictive performance models, thus neglecting performance-relevant information that is part of the software architecture. Approaches that extract architectural structures exist. However, the extraction then uses dynamic and static analyses in a controlled environment and in an offline setting, requiring manual instrumentation to obtain, e.g., the application control flow and data flow.

4. Architecture-Level Performance Abstractions for Online Use

Modern enterprise systems often have distributed application architectures composed of many independent services running in a heterogeneous environment (Papazoglou et al., 2007). In such systems, the applications are customized and new services are composed and deployed on-the-fly subjecting the system resources to varying workloads. Moreover, existing services, given their loosely-coupled nature, can evolve independently of one another. Managing the end-to-end application performance in such environments, i.e., satisfying service performance objectives while utilizing system resources efficiently, requires to answer questions such as:

- What performance would a new service or application deployed in the system environment exhibit and how much resources should be allocated to it?
- How should the workloads of the new service/application and existing services be partitioned among the available resources so that performance requirements are satisfied and resources are utilized efficiently?
- What would be the performance impact of adding a new component or upgrading an existing component as services and applications evolve?
- If an application experiences a load spike or a change of its workload profile, how would this affect the system performance? Which parts of the system architecture would require additional resources?
- What would be the effect of migrating a service or an application component from one server to another?

Answering such questions requires the ability to predict *online at system run-time* the performance impact of system configuration changes or workload changes. We refer to this as *online performance prediction.* In general, requirements for online performance prediction mechanisms are:

- The prediction mechanism should support metrics such as average resource utilization, service response time and service throughput. The average resource utilization is of interest for processing resources (e.g., CPUs) as well as for software resources (e.g., thread pools). For a service response time, estimating the mean, variance and distribution should be supported. The distribution is used to derive percentiles such as the 90th percentile, indicating an expected response time level for 90% of the requests. Since "the 90% percentile response time is closer to what a user would perceive in reality" (Liu, 2009), such percentiles are common metrics to reflect end-user performance. Service throughput is of interest when analyzing closed workloads, i.e., if the system workload is defined by a number of concurrent users and their think times.

- The prediction mechanism should support predicting the performance impact of changing service compositions. Service compositions include deploying or removing a service, replacing a service with another service implementation, and changes of how the services are connected to each other.
- The prediction mechanism should support predicting the performance impact of changes of resource allocations and system reconfigurations. Resource allocation is the assignment of resources to software components. System reconfigurations include adding or removing physical or virtual machines, and changing performance-influencing system parameters such as thread pool sizes.
- The prediction mechanism should support predicting the performance impact of different load-intensity levels and usage profiles. The load-intensity level is defined either by the inter-arrival time of user requests or by the number of concurrent users and their think times. The usage profile captures the services that are called, the order in which they are invoked, and the input parameters passed to them.
- The trade-off between prediction accuracy and time-to-result should be configurable. An accurate fine-grained performance prediction comes at the cost of higher prediction overhead and a longer prediction process. By using more coarse-grained performance models one can speed up the prediction process.
- The prediction accuracy must be adequate to support reasoning about service compositions, resource allocations and system reconfigurations with the goal to increase resource efficiency. According to Menasce and Virgilio (2000), for capacity planning a prediction error of 30% concerning mean response times and 5% concerning resource utilization is considered acceptable.

Existing architecture-level performance models provide a powerful tool for performance prediction at design-time (see Chapter 2). However, as described in Section 2.4, there are fundamental differences between offline and online scenarios for performance prediction. This leads to different design decisions concerning the underlying performance abstractions of the application architecture and concerning the respective performance prediction techniques. In particular, the type and amount of data available as a basis for model parameterization and calibration/adjustment at system design-time versus run-time is different. Furthermore, current approaches to modeling the component context in architecture-level performance models are not suitable for use at run-time since they do not provide enough flexibility in the way parameter and context dependencies can be expressed and resolved.

In this chapter, we propose new architecture-level performance abstractions for use in online scenarios (Brosig et al., 2013b, 2012). This involves: (i) a new approach to model performance-relevant service behavior at different levels of granularity, (ii) a new approach to parameterize performance-relevant properties of software components, and (iii) a new approach to model dependencies between parameters, specifically for use at run-time.

The presented modeling abstractions are part of the Descartes Modeling Language (DML), a new modeling language for run-time performance and resource management of modern dynamic IT service infrastructures. Figure 4.1 gives a high-level overview of DML. The system architecture meta-model consists of the application architecture meta-model and

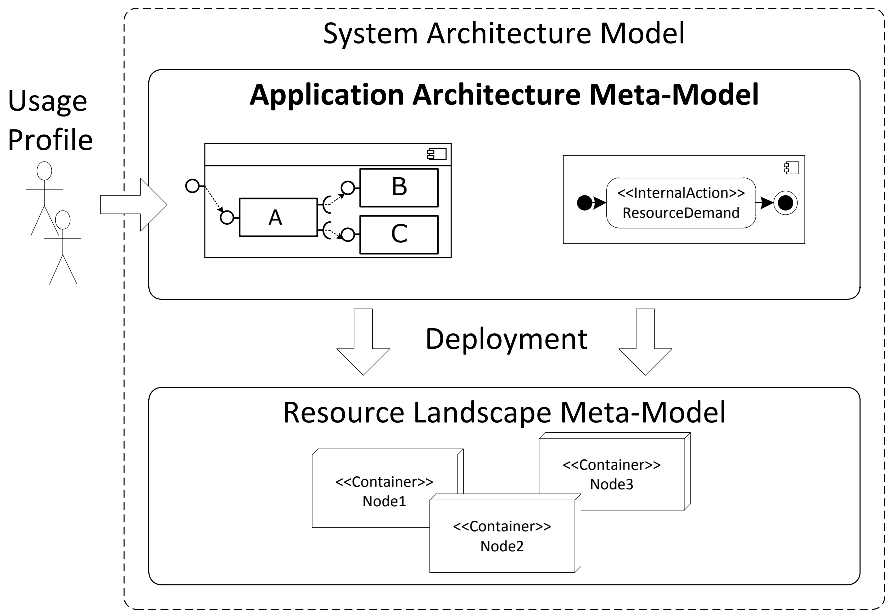

System Architecture Model
Usage Profile
Application Architecture Meta-Model
A
B
C
<<InternalAction>> ResourceDemand
Deployment
Resource Landscape Meta-Model
<<Container>> Node1
<<Container>> Node2
<<Container>> Node3

Figure 4.1: Structure of the Descartes Modeling Language (DML)

the resource landscape meta-model. The resource landscape meta-model allows modeling the physical and logical resources (e.g., virtualization and middleware layers) provided by modern dynamic data centers (Huber et al., 2012a). The application architecture meta-model allows modeling the performance-relevant service behavior of the applications executed in the resource landscape (Brosig et al., 2013b, 2012). These two meta-models are connected with the deployment meta-model, which can be used to describe how software components are deployed. The usage profile meta-model can be used to describe how users access the hosted applications.

In addition, DML provides means to define so-called adaptation points to describe the parts of the system architecture that are adaptable at run-time. The adaptation points span the configuration space of the modeled system. They describe the valid states a system can have at run-time. Furthermore, DML can be used to describe adaptation processes (Huber et al., 2012b, 2013). The language is intended to describe scenarios such as autonomic performance and resource management at run-time.

The remainder of this chapter is organized as follows: Section 4.1 introduces the application architecture meta-model in detail. Section 4.2 shortly describes the resource landscape and deployment meta-model. In Section 4.3, we introduce the usage profile meta-model. Section 4.4 summarizes this chapter.

4.1 Application Architecture

The application architecture is modeled as a component-based software system. The performance behavior of such a system is a result of the assembled components' performance behavior. In order to capture the behavior and resource consumption of a component, its behavior abstractions have to be described.

The application architecture meta-model is described in several subsections. Subsection 4.1.1 describes the underlying component model. Subsection 4.1.2 introduces a running example that is used to motivate and illustrate the new modeling concepts. Subsection 4.1.3 introduces novel service behavior abstractions. In Subsection 4.1.4, we present how the behavior abstractions are parameterized and in Subsection 4.1.5 we describe how

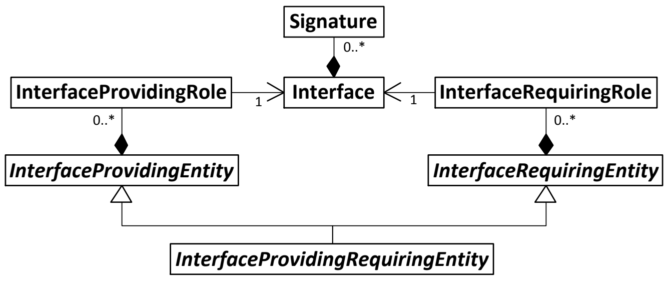


Figure 4.2: Components and Interfaces, cf. Becker et al. (2009)

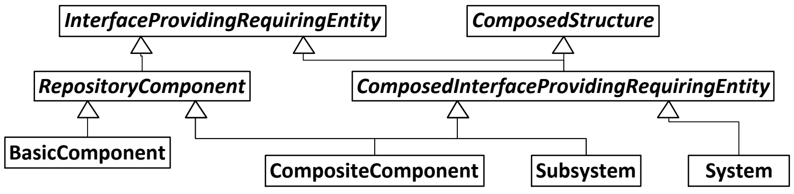


Figure 4.3: Component Type Hierarchy, cf. Becker et al. (2009)

we model probabilistic parameter dependencies specifically for use at run-time. An interface used to obtain empirical characterizations of model parameters from monitoring statistics is described in Subsection 4.1.6.

4.1.1 Component Model and System Model

The component model stems from the Palladio Component Model (PCM) (Becker et al., 2009; Reussner et al., 2011). Software building blocks are modeled as components. In the following, we describe how components are associated with interfaces they provide or require, and how composite components can be assembled from other components.

Components and interfaces are modeled as separate model entities, i.e., components as well as interfaces are first-class entities that can exist on their own. Consequently, a component does not *contain* an interface, but it may provide and/or require some interfaces (Szyperski et al., 2002). The connection between components and interfaces is specified using so-called *roles* (Becker et al., 2009). A component can take two roles relative to an interface. It can either provide and implement the functionality specified in the interface or it can require that functionality. Figure 4.2 shows the corresponding meta-model. An InterfaceProvidingEntity may have InterfaceProvidingRoles that refer to an Interface consisting of one or more method Signatures. An InterfaceRequiringEntity is modeled accordingly with InterfaceRequiringRoles. We refer to each method provided by a component as a service. We thus refer to the methods of the provided interfaces of a component as provided services, and refer to the methods of the required interfaces of a component as required services or external services.

An InterfaceProvidingRequiringEntity is the supertype of different component types that are shown in Figure 4.3. Basically, two types of components are distinguished. BasicComponents, i.e., atomic components, and CompositeComponents both can require and provide interfaces, and are stored in a component repository. Thus, they are subtypes of InterfaceProvidingRequiringEntity and RepositoryComponent. A CompositeComponent also inherits from type ComposedStructure, indicating that it is composed of other components. A Subsystem is similar to a CompositeComponent, but treated differently when it comes to modeling the deployment of components. While a CompositeComponent is deployed as a whole, the Subsystem is deployed by deploying all its child components. A System is

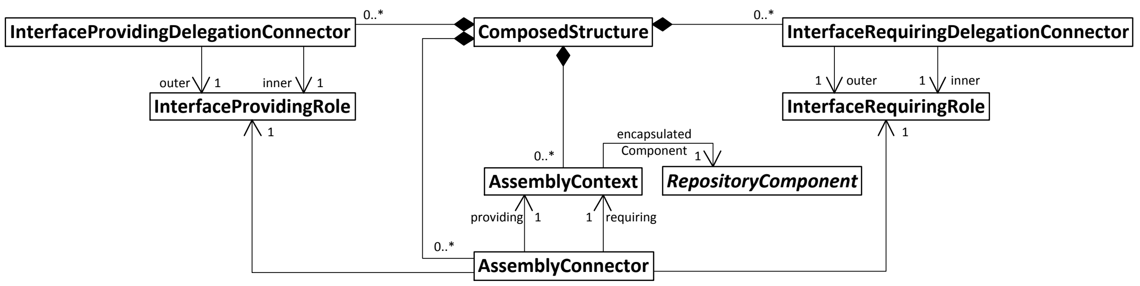


Figure 4.4: Component Composition, cf. Becker et al. (2009)

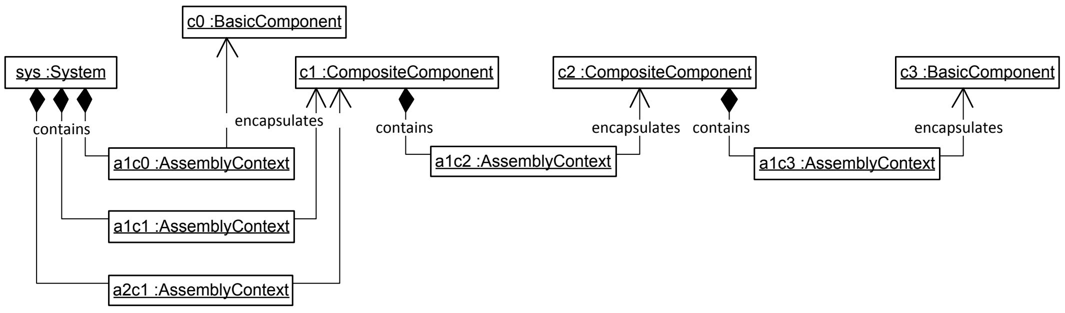


Figure 4.5: Example: System Instance as UML Object Diagram

similar to a CompositeComponent, the difference is that a System is not part of a component repository but a unique designated ComposedStructure. A System is the outermost ComposedStructure representing the system boundary.

Figure 4.4 shows how a ComposedStructure is assembled. A ComposedStructure may contain several AssemblyContexts which themselves each refer to a RepositoryComponent (referring to a Subsystem is only allowed if the parent ComposedStructure is of type System or Subsystem). Each AssemblyContext thus represents a child component instance in the composite. An AssemblyConnector connects two such child component instances with an InterfaceRequiringRole and an InterfaceProvidingRole, representing a connection between a providing role of the first component and a requiring role of the second component. Connectors from a child component instance to the composite component boundary are modeled using delegation connectors (InterfaceProvidingDelegationConnector and InterfaceRequiringDelegationConnector). The delegation connectors refer to a role of an inner child component instance and to a role of the outer ComposedStructure.

Figure 4.5 shows an exemplary instantiation of a ComposedStructure. It shows a System model as UML object diagram. The system instance *sys* contains three AssemblyContexts. Two of them refer to the same CompositeComponent *c1*, one refers to BasicComponent *c0*. Component *c1* contains an AssemblyContext *a1c2* that refers to another CompositeComponent *c2* that itself encapsulates BasicComponent *c3* via AssemblyContext *a1c3*. Figure 4.6 shows the same instance as component diagram. The outermost box represents System *sys*. AssemblyContext *a1c0* is connected to AssemblyContexts *a1c1* and *a2c1*, e.g., it could balance the load between the two instances of CompositeComponent *c1*.

Although there are only one AssemblyContext for component *c2* and only one AssemblyContext for component *c3*, the system diagram in Figure 4.6 illustrates that both components must be instantiated twice, because there are two instances of their surrounding component *c1*. Thus, an AssemblyContext is not equivalent to a component instance. An AssemblyContext is only unambiguous within its direct parent composite structure.

An AssemblyContext refers to a component type. This allows modeling different instances

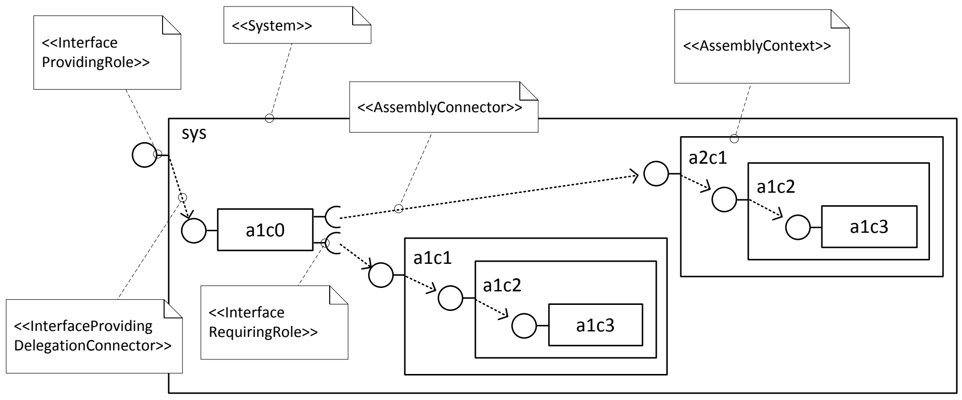

<<Interface ProvidingRole>>
<<System>>
<<AssemblyContext>>
<<AssemblyConnector>>
sys
a2c1
a1c2
a1c3
a1c0
a1c1
a1c2
a1c3
<<InterfaceProviding DelegationConnector>>
<<Interface RequiringRole>>

Figure 4.6: Example: System Instance

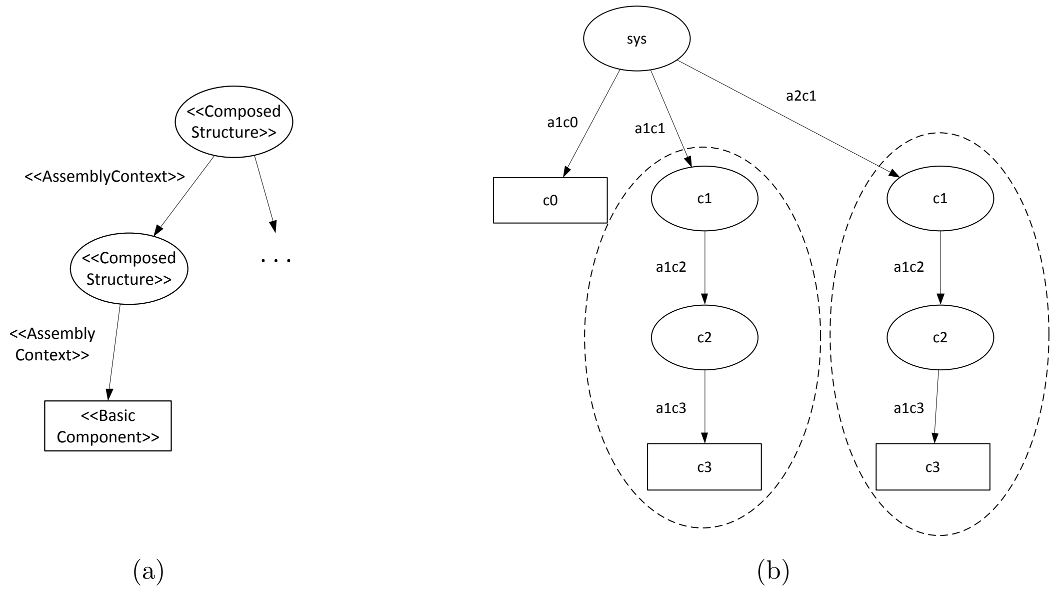


Figure 4.7: (a) Composition Tree Schema and (b) Example System Instance as Composition Tree

of the same component type, but in order to uniquely identify such an instance, an AssemblyContext is not sufficient. However, each component instance can be uniquely identified by a sequence of AssemblyContexts.

To illustrate the composition hierarchy of a ComposedStructure instance, we describe it as a *composition tree* $G = (V, E)$ where V is a set of nodes and E is a set of ordered pairs (v, v') with $v, v' \in V$, representing the set of links between the nodes.

- Each tree node $v \in V$ represents a component instance, denoted as $instance(v)$.
- $\forall v \in V$:
  $(\exists e = (v, v') \in E$
  $\Longleftrightarrow$
  $instance(v)$ has a child AssemblyContext that encapsulates $instance(v'))$

The inner nodes of G represent instances of ComposedStructures, the leaves of G represent instances of BasicComponents. Figure 4.7(a) illustrates such a composition tree.

In a ComposedStructure *cs*, a component instance is uniquely identified by a path from the node representing *cs* to the node representing the specific component instance. Figure 4.7(b) shows the example of Figure 4.5 as composition tree. The root of the tree is

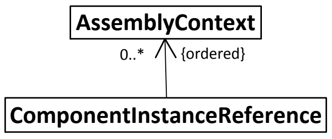


Figure 4.8: Component Instance Reference

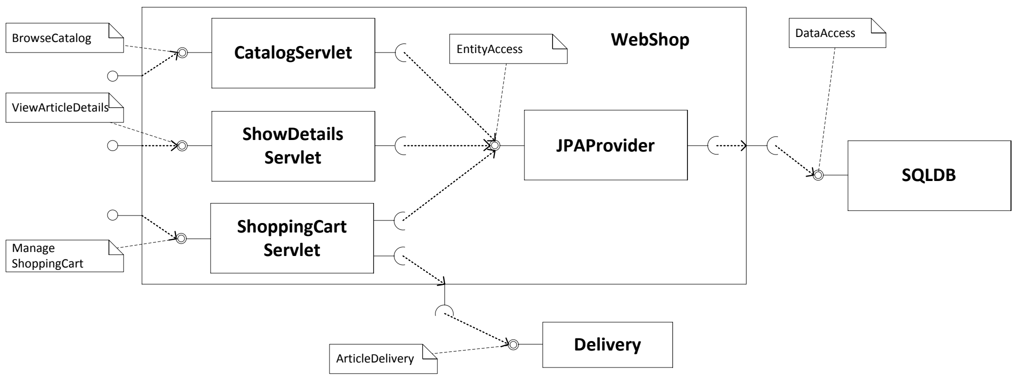


Figure 4.9: Running Example: WebShop

System *sys*. The two instantiations of CompositeComponent *c1* are highlighted with dashed lines. In the example, the two instances of BasicComponent *c3* can thus be identified by the path {*a1c1*, *a1c2*, *a1c3*} and the path {*a2c1*, *a1c2*, *a1c3*}, respectively.

A ComponentInstanceReference can thus be modeled as a sequence of AssemblyContexts as shown in Figure 4.8. The difference between a component type and its instances is of relevance in Section 4.1.4 where we distinguish between parameterizations at the component type level and at the component instance level.

4.1.2 Running Example

We introduce a running example to illustrate the novel modeling abstractions we propose in the following sections. Figure 4.9 shows a simple online shop, consisting of a *WebShop* composite component, and an *SQLDB* component. The *WebShop* consists of several Java Servlet components, the entity data is accessed using a Java Persistence API (JPA) provider component (*JPAProvider*). The *CatalogServlet* allows browsing the catalog of available articles. The *ShowDetailsServlet* implements a view of the article details. The *ShoppingCartServlet* provides a shopping cart including payment processing and requires an external *ArticleDelivery* service that is implemented by the *Delivery* component.

4.1.3 Service Behavior Abstractions

This section introduces different service behavior abstraction levels. Section 4.1.3.1 provides the motivation for the new service behavior abstractions, Section 4.1.3.2 describes the modeling approach and Section 4.1.3.3 describes the corresponding meta-model in detail. In Section 4.1.3.4, we provide an illustrative example.

4.1.3.1 Motivation

In order to ensure Service Level Agreements (SLAs) while at the same time optimizing resource utilization, the service provider needs to be able to predict the system performance under varying workloads and dynamic system reconfigurations. The underlying performance models enabling online performance prediction must be parameterized and

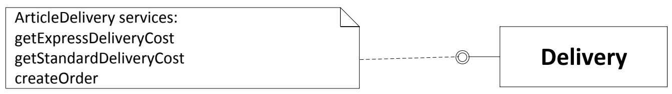


Figure 4.10: Example: *Delivery* Component

analyzed on-the-fly. Such models may be used in many different scenarios with different requirements for accuracy and timing constraints. Depending on the time horizon for which a prediction is made, online models may have to be solved within seconds, minutes, hours, or days, and the same model should be usable in multiple different scenarios with different requirements for prediction accuracy and analysis overhead. Hence, in order to provide flexibility at run-time, our meta-model must be designed to support multiple abstraction levels and different analysis techniques allowing to trade-off between prediction accuracy and time-to-result.

Explicit support for multiple abstraction levels is also necessary since we cannot expect that the monitoring data needed to parameterize the component models would be available at the same level of granularity for each system component. For example, even if a fine granular abstraction of a component behavior is available, depending on the platform on which the component is deployed, some model parameters might not be resolvable at run-time, e.g., due to the lack of monitoring capabilities allowing to observe the component's internal behavior. In such cases, it is inevitable to use a more coarse-grained abstraction of the component behavior that only requires observing the component's behavior at the component boundaries.

In the following, we describe three practical examples where models at different abstraction levels are needed, based on the *WebShop* example introduced in Figure 4.9. The *Delivery* component provides services to calculate the cost of a delivery and to create a delivery order (see Figure 4.10). Two kinds of deliveries are supported: a standard delivery and an express delivery.

Assume that the *Delivery* component is an outsourced service hosted by a different service provider, the only type of monitoring data that would typically be available for the *createOrder* service is response time data. In such a case, information about the component-internal behavior or resource consumption would not be available and, from the perspective of our system model, the component would be treated as a "black-box".

If the *Delivery* component is a third party component hosted locally in our environment, monitoring at the component boundaries including measurements of the resource consumption as well as external calls to other components would typically be possible. Such data allows to estimate the resource demands of each provided component service (e.g., using techniques presented in Section 6.4.3) as well as frequencies of calls to other components. Thus, in this case, a more detailed model of the component can be built, allowing to predict its response time and resource utilization for different usage scenarios.

Finally, if the internal behavior of the *Delivery* component including its control flow and resource consumption of internal actions can be monitored, more detailed models can be built allowing to obtain more accurate performance predictions including response time distributions. Predicting response time distributions is relevant for example in situations where SLAs with service response time limits defined in terms of response time percentiles need to be evaluated.

In summary, it is important to support the modeling of service behavior at different levels of abstraction and detail. The models should be usable in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to accurate performance prediction. Furthermore, the modeled abstraction

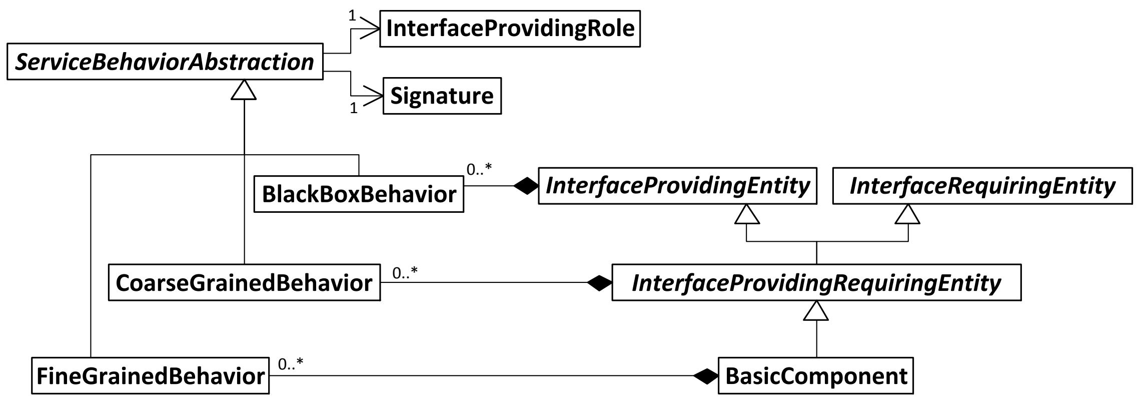


Figure 4.11: Different Service Behavior Abstractions

level depends on the information that monitoring tools can obtain at run-time, e.g., to what extent component-internal information is available.

4.1.3.2 Modeling Approach

To provide maximum flexibility, for each provided service, our proposed meta-model supports having multiple (possibly co-existing) behavior abstractions at different levels of granularity:

- **Black-box behavior abstraction.** A "black-box" abstraction is a probabilistic representation of the service response time behavior. Resource demands are not specified. This representation captures the view of the service behavior from the perspective of a service consumer without any additional information about the service's behavior.
- **Coarse-grained behavior abstraction.** A "coarse-grained" abstraction captures the service behavior when observed from the outside at the component's boundaries. It consists of a description of the frequency of external service calls and the overall service resource demands. Information about the service's total resource consumption and information about external calls made by the service is required, however, no information about the service's internal control flow is assumed.
- **Fine-grained behavior abstraction.** A "fine-grained" abstraction captures the performance-relevant service control flow which is an abstraction of the actual control flow. Performance-relevant actions are component-internal computational tasks, the acquisition and release of locks, as well as external service calls, thus also loops and branches where external services are called. Furthermore, the ordering of external service calls and internal computations may have an influence on the service performance. The control flow is modeled at the same abstraction level as the Resource Demanding Service Effect Specification (RDSEFF) of PCM (cf. Becker et al. (2009)), however, there are significant differences in the way model variables and parameter dependencies are modeled. The details of these are presented in Section 4.1.4 and Section 4.1.5. In contrast to the coarse-grained behavior description, a fine-grained behavior description requires information about the internal performance-relevant service control flow including information about the resource consumption of internal service actions.

4.1.3.3 Modeling Abstractions

Figure 4.11 shows the meta-model elements describing the three proposed service behavior abstractions. Type FineGrainedBehavior is attached to the type BasicComponent, a com-

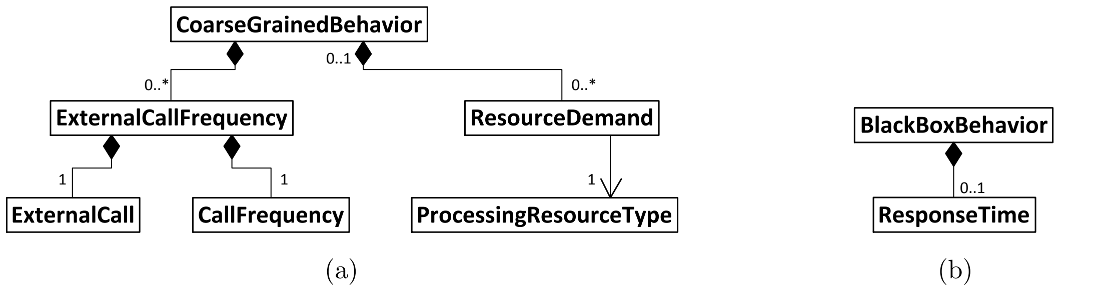


Figure 4.12: (a) Coarse-Grained and (b) Black-Box Behavior Abstractions

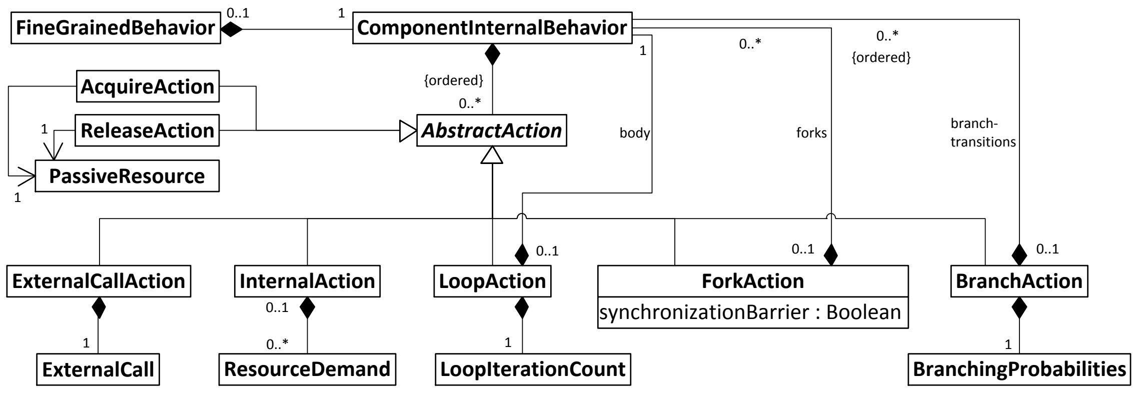


Figure 4.13: Fine-Grained Behavior Abstraction, cf. Becker et al. (2009)

ponent type that cannot contain further subcomponents. The CoarseGrainedBehavior is attached to type InterfaceProvidingRequiringEntity that generalizes the types System, Subsystem, CompositeComponent and BasicComponent. Type BlackBoxBehavior is attached to type InterfaceProvidingEntity, neglecting external service calls to required services. Thus, in contrast to the fine-grained abstraction level, the coarse-grained and black-box behavior descriptions can also be attached to service-providing *composites*, i.e., ComposedStructures.

The meta-model elements for the CoarseGrainedBehavior and BlackBoxBehavior abstractions are shown in Figure 4.12. A CoarseGrainedBehavior consists of ExternalCallFrequencies and ResourceDemands. An ExternalCallFrequency characterizes the type and the number of external service calls. Type ResourceDemand captures the total service time required from a given ProcessingResourceType. A ProcessingResourceType is, e.g., a CPU, HDD or network. A BlackBoxBehavior, on the other hand, can be described with a ResponseTime characterization.

Figure 4.13 shows the meta-model elements for the fine-grained behavior abstraction. A ComponentInternalBehavior models the abstract control flow of a service implementation. Calls to required services are modeled using so-called ExternalCallActions, whereas internal computations within the component are modeled using InternalActions, characterized with ResourceDemands. Access to PassiveResources with semaphore semantics (e.g., thread pools) can be modeled via AcquireAction to obtain the resource and ReleaseAction to release the resource. Nested control flow actions like LoopAction, BranchAction or ForkAction are only used when they affect calls to required services (e.g., if a required service is called within a loop, a corresponding LoopAction is modeled; otherwise, the whole loop would be captured as part of an InternalAction). The nested control flow actions contain further ComponentInternalBehavior models, either as loop body, as forks, or as branch transitions. LoopActions and BranchActions can be characterized with loop iteration counts and branching probabilities, respectively. ForkActions can be modeled either with or without a synchronization barrier. A barrier for the group of the ForkAction's forks means that the

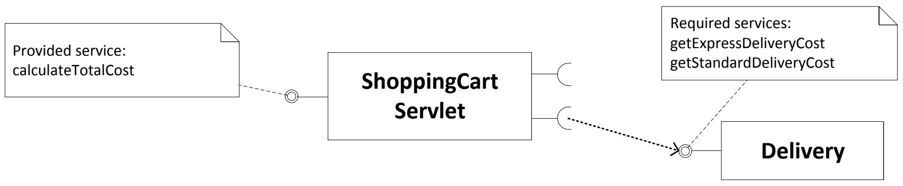


Figure 4.14: Example: *Delivery* and *ShoppingCartServlet*

control flow only proceeds when all forks have reached the barrier.

Note that it is prohibited to model *cycles* of services, i.e., a service requiring external services that themselves require that service.

Furthermore, a service can be modeled at different behavior abstraction levels, e.g., a service can be described using both a coarse-grained abstraction and a black-box abstraction. However, if a service is described by both a fine-grained behavior and a coarse-grained behavior, the following must hold: The set of processing resource types of all resource demands of the coarse-grained behavior must be a subset of the processing resource types of all resource demands of the fine-grained behavior, i.e., all resource types used by the coarse-grained behavior must also be used by the fine-grained behavior, if both behavior descriptions exist. Furthermore, the set of called external services of the coarse-grained behavior must be a subset of the called external services of the fine-grained behavior, i.e., external service calls modeled in the coarse-grained behavior description must also be modeled in the fine-grained description, if both behavior descriptions exist.

4.1.3.4 Example

In the *WebShop* example, the *ShoppingCartServlet* provides a service called *calculateTotalCost* that calculates the cost of all items in the shopping cart including delivery costs (see Figure 4.14). To obtain the delivery costs depending on the user preferences, the *ShoppingCartServlet* either calls service *getExpressDeliveryCost* or service *getStandardDeliveryCost*. These two services are provided by the *Delivery* component. In this example, the probability of standard delivery is 0.8, and 0.2 for express delivery.

A fine-grained model of service *calculateTotalCost* is depicted in Figure 4.15. The service behavior reflects the service's internal control flow. There is a branch action that either leads to an external service call to *getStandardDeliveryCost* or an external service call to *getExpressDeliveryCost*. The branching probabilities are annotated accordingly, with 0.8 for the first branch transition, and 0.2 for the second branch transition. Note that the annotation `EnumPMF[(‘Branch1’;0.8)(‘Branch2’;0.2)]` is explained in Section 4.1.4.

A coarse-grained model of service *calculateTotalCost* is depicted in Figure 4.16. The external service calls to *getStandardDeliveryCost* and *getExpressDeliveryCost* are modeled as they can be observed from the component boundary of component *ShoppingCartServlet*. For each call to *calculateTotalCost*, a respective external service is either called once or not at all. An external call to *getExpressDeliveryCost* has a frequency of 1 with a probability of 0.2, and 0.8 otherwise. For external call *getStandardDeliveryCost*, the probabilities are vice versa. The annotations of the form `IntPMF[(0;0.2)(1;0.8)]` are explained in Section 4.1.4. However, the exclusive relationship between the two external service calls is not reflected in the coarse-grained model. This leads to deviations when deriving the response time distribution of service *calculateTotalCost*.

4.1.4 Parameterization

This section introduces the parameterization concept of the service behavior abstractions described in the previous section. Section 4.1.4.1 provides the rationale, Section 4.1.4.2 de-

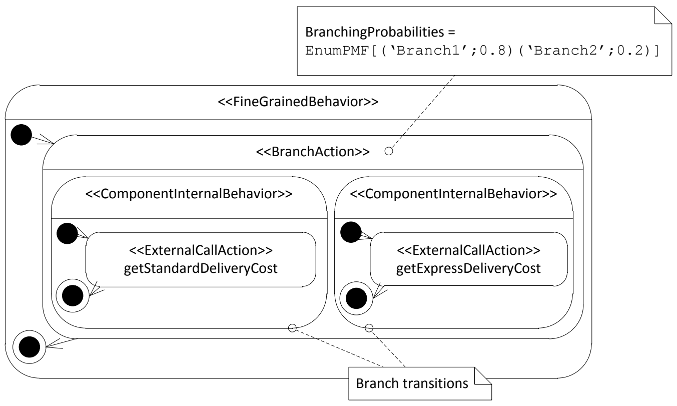


Figure 4.15: Example: Fine-Grained Behavior Abstraction of Service *calculateTotalCost* Provided by *ShoppingCartServlet*

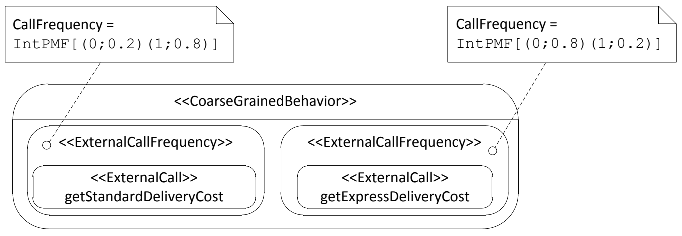


Figure 4.16: Example: Coarse-Grained Behavior Abstraction of Service *calculateTotalCost* Provided by *ShoppingCartServlet*

scribes the modeling approach and Section 4.1.4.3 describes the corresponding abstractions in detail. In Section 4.1.4.4, we provide an illustrative example.

4.1.4.1 Motivation

The behavior abstractions described in Section 4.1.3 have to be parameterized with resource demands, response times, frequencies of external calls, loop iteration counts and branching probabilities. In the context of online performance prediction, these parameters are typically characterized based on monitoring data collected at run-time. The measurements are gathered at component *instance* level. Thus, the question arises if the measurements, e.g., branching probabilities collected for an instance of a certain component type, are representative for the corresponding branching behavior at another instance of the same component type.

Assume the *WebShop* introduced in Section 4.1.2 is instantiated for different stores. For example, one instance of the web shop serves as an online supermarket and another instance of the web shop serves as a game store. See Figure 4.17 for an illustration. Technically, the component implementation is the same, but the performance behavior may differ among the two instances. One reason is the different usage behavior. While a supermarket client typically buys many items at once, a client of the game store typically buys only few items per order. Another reason is the different article database. The game store may provide video sequences when showing article details, the supermarket may only show static article images. When parameterizing a performance model, measurements of, e.g., *ShowDetailsServlet*, are likely to differ between the two shops. Although the shops use the same component types, the underlying shop data is different. In enterprise software systems, model parameters depending on the state of the database are common (Fowler, 2002; Rolia and Vetland, 1995). However, it is not appropriate to model this data dependency explicitly, because it depends on internals of the database system.

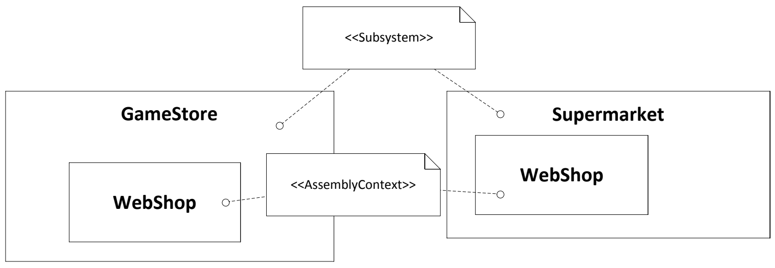


Figure 4.17: Example: WebShops for a *GameStore* and a *Supermarket*

Whether the characterization of a model parameter is valid across component boundaries thus depends on the specific considered parameter. For the same component type, there can be parameters that are different for each component instance, or there might be model parameters that can be treated as identical across all instances of the component type. This is in contrast to design-time models such as PCM where representative monitoring data is typically not available to distinguish such cases.

4.1.4.2 Modeling Approach

In order to tackle different characterizations of model parameters, we provide means to specify so-called *scopes* of model parameters explicitly. A *scope* of a model parameter specifies a context where the parameter is unique. This means, on the one hand, that measurements of the parameter can be used interchangeably among component instances provided that these instances belong to the same scope. On the other hand, it means that measurements of the parameter are not transferable across scope boundaries. Thus, if monitoring data for a given parameter is available, it should be clear based on its scope for which other instances of the component this data can be reused.

If for a given model parameter of a component, the component developer is aware that corresponding monitoring statistics are not reusable across different instances of the component, the developer can define a scope for the model parameter to indicate that. Furthermore, the developer of a composite component can define the composite component's boundary as scope for a contained model parameter, thus restricting the usage of monitoring statistics for that model parameter to usage only within that composite component. Note that the developer of the composite component cannot widen an existing scope of a model parameter, but can restrict it to the composite's boundary. In the running example, for instance, the developer of composite component *WebShop* knows that different instances of the component represent different tenants and thus different underlying shop data, and hence can define composite component *WebShop* as scope of the involved model variables.

In case a model parameter does not have a specified scope, i.e., in the default case, the model parameter is globally reusable. Monitoring data from all observed instances of the component can then be used interchangeably and treated as a whole. Moreover, once a model parameter has been characterized empirically (e.g., "learned" from monitoring data), it can be used for all instances of the component in any current or future system.

4.1.4.3 Modeling Abstractions

Figure 4.18 shows the possible ModelVariables of service behavior abstractions that have to be parameterized. On the one hand, there are timing parameters such as ResourceDemand (for coarse-grained and fine-grained service behavior abstractions) and ResponseTime (for

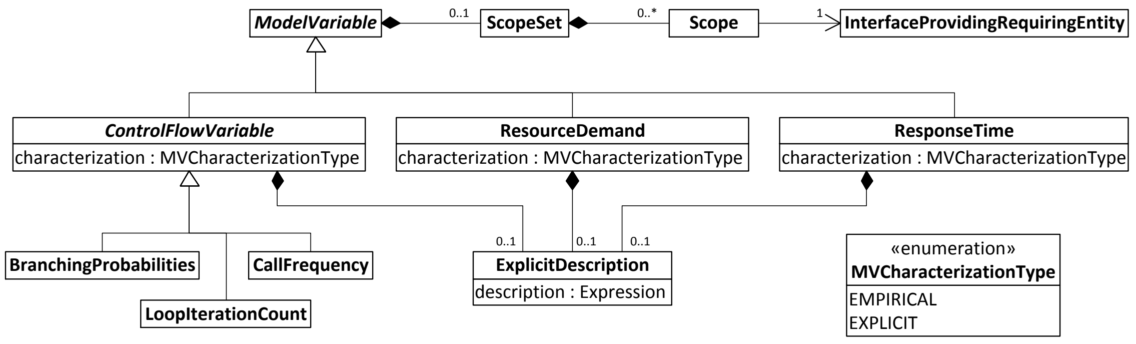


Figure 4.18: Model Variables

black-box service behavior abstractions). On the other hand, there are control-flow related parameters such as CallFrequency, LoopIterationCount and BranchingProbabilities.

There are two ways to characterize the mentioned model variables. Either EMPIRICAL (default) or EXPLICIT can be chosen as a characterization type. EMPIRICAL means that the model variable has to be quantified using monitoring statistics, i.e., it is characterized using empirical data that is accessed via an interface to the monitoring infrastructure. The interface is presented in Section 4.1.6. EXPLICIT means that the model variable is characterized explicitly. In this case, an ExplicitDescription can be used to specify a random variable by means of the Stochastic Expression (StoEx) language proposed by Koziolek (2008). The StoEx language allows characterizing discrete probability distributions with Probability Mass Functions (PMFs), approximating continuous probability densities with samples, or using common probability distribution functions such as the exponential distribution or the binomial distribution. Furthermore, the expression language allows specifying random variables “as a combination of several other random variables using arithmetic or boolean operations” (Koziolek, 2008, p.66).

The model variables LoopIterationCount and CallFrequency are discrete random variables defined on the sample space $\Omega = \mathbb{N}_0 = \mathbb{N} \cup \{0\}$. A typical PMF for a loop is described, e.g., with the expression `IntPMF[(9;0.2)(10;0.5)(11;0.3)]`. This PMF expressed as StoEx specifies that the loop body is executed 9 times with a probability of 0.2, 10 times with a probability of 0.5, and 11 times with a probability of 0.3. Model parameter BranchingProbabilities is also described with a discrete random variable, however, its sample space Ω is the set of branch transitions of the corresponding BranchAction. The branch transitions are ordered, thus we can use their indexes as identifiers. A PMF for the branching probabilities of a branch with two branch transitions is, e.g, `EnumPMF[(‘Branch1’;0.2)(‘Branch2’;0.8)]`, meaning that the transition with index 1 has a probability of 0.2, the transition with index 2 has a probability of 0.8.

The random variables of the remaining two model variables ResponseTime and ResourceDemand are typically defined on the sample space $\Omega = \mathbb{R}_{\geq 0}$, where $\omega \in \Omega$ is interpreted as timing value. They are described by a Probability Density Function (PDF) that is either approximated (see Koziolek (2008, p.67) for an illustration) or defined using common distributions such as the exponential distribution. An exemplary approximation $f_X^{'}$ for a density function $f_X(x)$ of a random variable X is `DoublePDF[(10.0;0.0)(30.0;0.04)(32.0;0.1)]`, meaning

$$f_X^{'}(x) = \begin{cases} 0.0 & x < 10.0, \\ 0.04 & 10 \leq x < 30.0, \\ 0.1 & 30 \leq x < 32, \\ 0.0 & 32 \leq x. \end{cases}$$

An exponential distribution (with $\lambda = 1$) as StoEx is denoted as `Exp(1)`.

As shown in Figure 4.18, a Scope is modeled as a reference to an InterfaceProvidingRequiringEntity. A ModelVariable may have several scopes, modeled as ScopeSet, because it may be assembled in different ComposedStructures. In the following, the semantics of such a ScopeSet is defined. Let v be a model variable, and c^v the (composite) component or (sub-)system where the model variable v resides. Let $S^v = \{s_1^v, \ldots, s_n^v\}$ denote the set of (composite) components or (sub-)systems referenced as Scope of v. Let $S_0^v = S^v \cup \{s_0\}$, where s_0 represents the system. An instance of c^v, as shown in Figure 4.8 in Section 4.1.1, can be identified as a sequence of AssemblyContexts $\{a_1, \ldots, a_m\}$. The container of AssemblyContext a_1 is the system, and *encapsulatedComponent*$(a_m) = c^v$ meaning that AssemblyContext a_m encapsulates c^v. We then define

$$\begin{aligned} &\mathit{evalscope}^v(\{a_1, \ldots, a_m\}) \\ &= \begin{cases} \{a_1, \ldots, a_j\} & , \begin{array}{l} \exists j \in \{1, \ldots, m\} : (\ \mathit{encapsulatedComponent}(a_j) \in S_0^v \wedge \\ \qquad \forall k \in \{j+1, \ldots, m\} : \mathit{encapsulatedComponent}(a_k) \notin S_0^v\ ) \end{array} \\ s_0 & , \text{ otherwise.} \end{cases} \end{aligned}$$

Measurements of model variable v at instance $\{a_1, \ldots, a_m\}$ are then valid within instance *evalscope*$^v(\{a_1, \ldots, a_m\})$. Function *evalscope*$^v(\{a_1, \ldots, a_m\})$ evaluates to a (composite) component or (sub-)system instance whose type is in the set S_0^v, namely the innermost instance when traversing from the instance identified by $\{a_1, \ldots, a_m\}$ to the system s_0. Note that the scope of a variable is only considered if EMPIRICAL is chosen as a characterization type.

4.1.4.4 Example

In the *WebShop* example, a ModelVariable v whose scope is set to the surrounding composite component *WebShop* is parameterized differently for each instance of component *WebShop*. Consequently, in Figure 4.17, the variable v has different values for the game store and the supermarket instances. If the scope is omitted, the value of variable v is shared across all component instances.

4.1.5 Probabilistic Parameter Dependencies

This section introduces the modeling concepts for describing probabilistic parameter dependencies. Section 4.1.5.1 provides the rationale, Section 4.1.5.2 describes the modeling approach and Section 4.1.5.3 describes the corresponding abstractions in detail. In Section 4.1.5.4, we provide an illustrative example.

4.1.5.1 Motivation

Figure 4.19 shows how the *CatalogServlet* component in the WebShop example is connected to the *JPAProvider* component. The servlet provides a service to list the articles of the shop. The list is normally composed of multiple pages. By providing the argument *pagenumber*, the user can choose which page to view. For each article in the list, the servlet shows a preview image of the article. Preview images are loaded via the JPA provider. The JPA provider itself issues a query to load a preview image, but only if the image is not already available in the JPA cache.

We are now interested in the probability of calling *issueNamedQuery_previewImage* because it results in a costly database access. As illustrated in the fine-grained service behavior in Figure 4.20, the probability of calling the database corresponds to a branch probability in the control flow of the *getArticlePreviewImage* service. This probability depends on

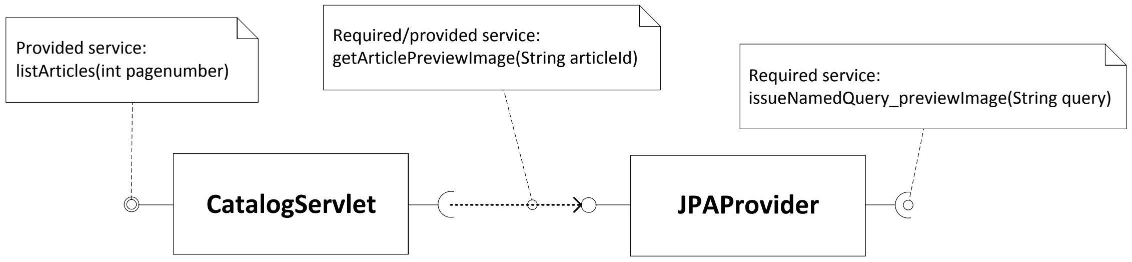

Provided service:
listArticles(int pagenumber)
Required/provided service:
getArticlePreviewImage(String articleId)
Required service:
issueNamedQuery_previewImage(String query)
CatalogServlet
JPAProvider

Figure 4.19: Example: *CatalogServlet* and *JPAProvider* Components

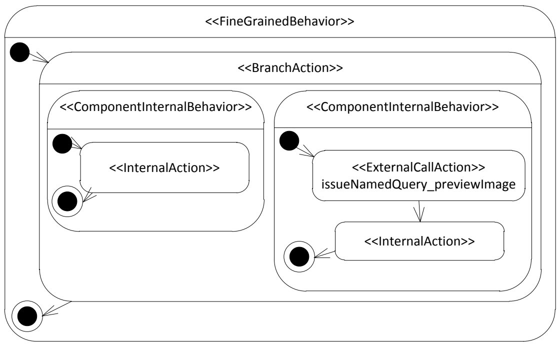


Figure 4.20: Example: Cache Miss or Cache Hit in Service *getArticlePreviewImage*

whether the article preview image is in the JPA cache or not. If the article (identified by parameter *articleId*) is viewed frequently, the probability of a database call is low compared to an article that is rarely shown to the users.

As discussed in Section 2.3, some architecture-level performance models for design-time analysis allow modeling dependencies of the service behavior (including branching probabilities) on input parameters passed over the service's interface upon invocation. However, in this case, the only parameter passed is *articleId*. Such a parameter does not allow modeling the dependency because the branching probabilities depend on the state of the JPA cache. Furthermore, the interface between the component and the cache is too generic to infer direct parameter dependencies. This state-dependency is typical for modern business information systems (Fowler, 2002; Rolia and Vetland, 1995). The behavior of components is often dependent on the state of data containers such as caches or on persistent data stored in a database. However, modeling the state of a cache and/or a database is extremely complex and infeasible to consider as part of the performance model. Thus, in such a scenario, the approach of providing explicit characterizations of parameter dependencies is neither applicable nor appropriate.

However, in the example illustrated in Figure 4.20, there is a dependency between the branching probabilities and the service input parameter *pagenumber* of service *listArticles* provided by *CatalogServlet*. Figure 4.21 shows the fine grained behavior abstraction of the *listArticles* service. There is a loop where for each article of the requested page the corresponding preview image is requested. Intuitively, one would assume the existence of the following dependency: The higher the page number of the article list to show, the higher the probability that the articles' preview images are not in the JPA cache. This dependency cannot be modeled using existing approaches such as PCM since, on the one hand, two separate components are involved, i.e., the *pagenumber* parameter is external to component *JPAProvider*, and on the other hand, an explicit characterization of the dependency by a function is impractical to obtain. In such a case, provided that the existence of the parameter dependency is known, monitoring statistics collected at

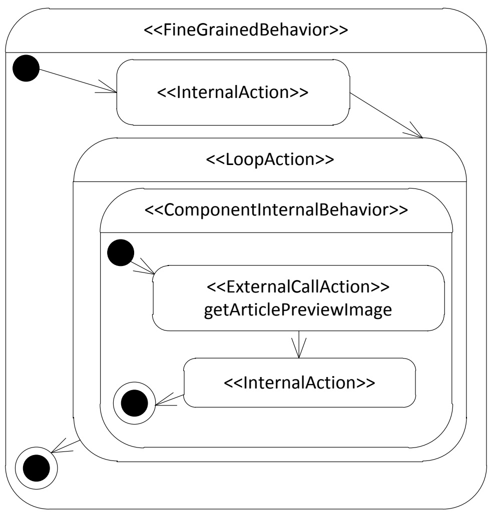


Figure 4.21: Example: Behavior of *listArticles* Service Provided by *CatalogServlet*

run-time can be used to characterize the dependency probabilistically. At run-time, the dependency between the values of influencing parameter *pagenumber* and the observed relative frequency of the *issueNamedQuery_previewImage* service calls can be monitored. Using this data, the branching probabilities in Figure 4.20 can be characterized more accurately as conditional probabilities, depending on values of parameter *pagenumber* of service *listArticles*.

The parameter dependency in the example can be considered typical for enterprise software systems. The behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation (Fowler, 2002; Rolia and Vetland, 1995). Such parameters are often not traceable directly over the service interface and tracing them requires looking beyond the component boundaries, e.g., the parameters might be passed to another component in the call path and/or they might be stored in a database structure queried by the invoked service. Furthermore, even if a dependency can be traced back to an input parameter of the called service, in many practical situations, providing an explicit characterization of the dependency is not feasible (e.g., using PCM's approach) and a probabilistic representation based on monitoring data is more appropriate. This situation is common in business information systems and our modeling abstractions must provide means to deal with it.

4.1.5.2 Modeling Approach

To allow the modeling of the above described parameter dependencies our architecture-level performance abstractions support the definition of so-called *influencing parameters*. In order to resolve parameter dependencies using monitoring data, such influencing parameters need to be mapped to some observable parameters that would be accessible at run-time.

Figure 4.22 illustrates our modeling approach in the context of the presented example from Figure 4.19. The branching probabilities of issuing a database query or not in the *getArticlePreviewImage* service are represented as *InfluencedVariable1*. The component developer is aware of the existence of the dependency between the branch probability and the *frequency* of the article to be listed. However, the developer does not have direct access to the *pagenumber* parameter of the *listArticle* service and does not know where the parameter might be observable and traceable at run-time. Thus, to declare the existence of the dependency, the component developer first defines an *InfluencingParameter1* named

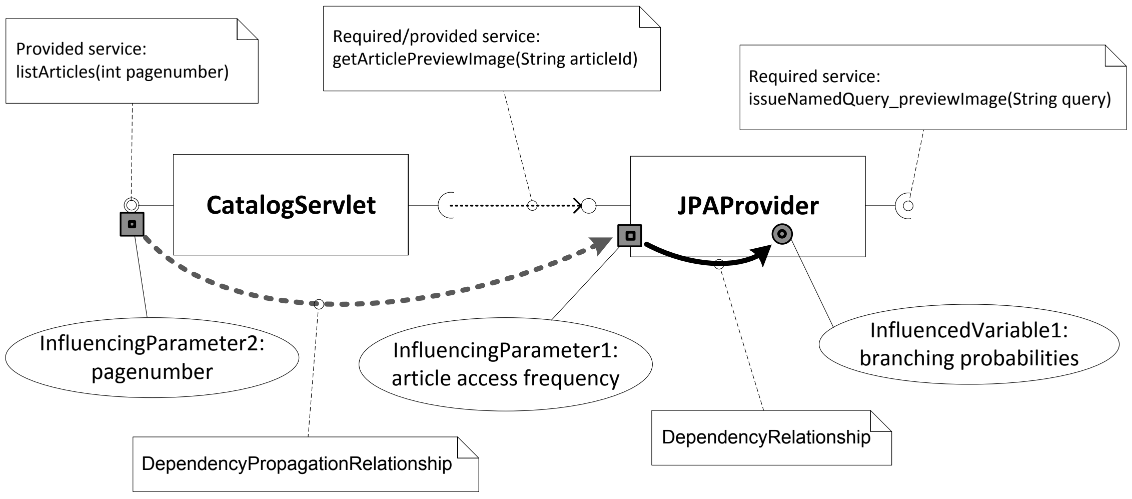


Figure 4.22: Modeling Parameter Dependencies

article access frequency, representing a so-called *shadow parameter*, and provides a textual description of that parameter's semantics. Parameter *article access frequency* characterizes how frequent the article with id *articleId* is accessed. The developer can then declare a *dependency* relationship between *InfluencedVariable1* and *InfluencingParameter1*.

The developer of composite component *WebShop* is then later able to link *Influencing-Parameter1* to the respective service input parameter *pagenumber* of the *CatalogServlet* component, designated as *InfluencingParameter2*. He does not explicitly know *how* the values of the page number relate to *InfluencingParameter2*, but he assumes that there is a dependency. We refer to such a link as declaration of a *dependency propagation* relationship between two influencing parameters. Having specified the influenced variable and the influencing parameters, as well as the respective dependency and dependency propagation relationships, the parameter dependency can then be characterized empirically. Our modeling approach supports both empirical and explicit characterizations for both dependency and dependency propagation relationships between model variables.

Note that an influencing parameter does not have to belong to a provided or required interface of the component. It can also be an auxiliary model entity allowing to model parameter dependencies in a flexible way. In that case, the influencing parameter is denoted as shadow parameter.

If an influencing parameter cannot be observed at run-time, the component's execution is obviously not affected, however, the parameter's influence cannot be taken into account in online performance predictions. The only thing that can be done in such a case is to monitor the influenced variable independently of any influencing factors and treat it as an invariant. It is important to note that parameter dependencies are intended to improve the prediction accuracy when considering parameter variations, however, if they cannot be characterized empirically using monitoring data, a performance prediction can still be conducted. To provide maximum flexibility, it is also possible to map the same influencing parameter to multiple other influencing parameters, some of which might not be monitorable in the execution environment, others could be monitorable with different overhead. Depending on the availability of monitoring data, some of the defined mappings might not be usable in practice and others could involve different monitoring overhead. Given that the same mapping can be usable in certain situations and not usable in others, the more mappings are defined, the higher flexibility is provided for resolving context dependencies at run-time.

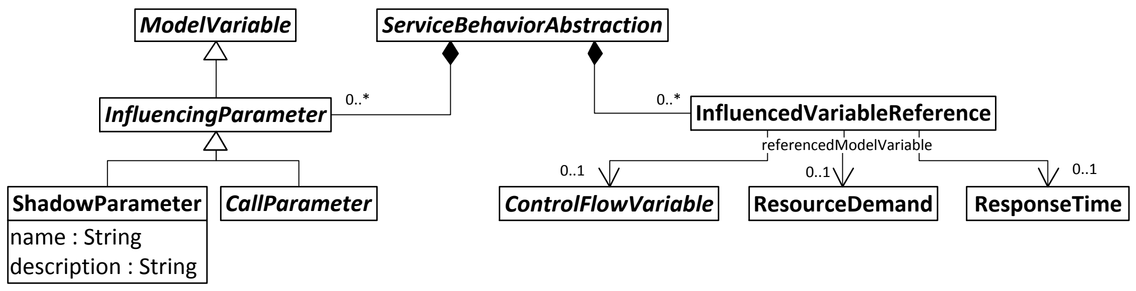


Figure 4.23: Influenced Variables and Influencing Parameters

In the following, we present the meta-model elements for influenced variables, influencing parameters, dependencies, and dependency propagations in more detail.

4.1.5.3 Modeling Abstractions

We first describe influenced variables and influencing parameters. Then we present the modeling abstractions to model the different types of relationships between model variables as they are shown in Figure 4.22. Then we describe how we characterize the introduced relationships.

Influenced Variables and Influencing Parameters

As shown in Figure 4.23, an influenced variable is indicated by an InfluencedVariableReference, referring to either a ControlFlowVariable, a ResourceDemand, or a ResponseTime.

An InfluencingParameter represents a parameter that has an *influence on* other model variables. Such parameters are either CallParameters or ShadowParameters, modeled as subtypes of InfluencingParameter.

A CallParameter, see Figure 4.24, is either a service input parameter, an external call parameter, or a return parameter of an external call. Given that in performance models, a service call parameter is only modeled if it is performance-relevant (see Figure 4.25), each modeled service call parameter can be considered to have a performance influence. Furthermore, the proposed modeling abstractions support referring not only to a parameter *VALUE*, but also to other characterizations such as NUMBER_OF_ELEMENTS if the referenced data type is a collection (cf. Becker et al. (2009)).

A ShadowParameter is an InfluencingParameter with a designated name and description. These attributes are intended to provide a human-understandable description that could be used by component developers, system architects, or performance engineers to identify and model relationships between model variables. A ShadowParameter can be considered as an auxiliary model entity allowing to model parameter dependencies in a flexible way. In order to resolve parameter dependencies using monitoring data, ShadowParameters need to be mapped to some observable parameters that are accessible at run-time.

Furthermore, note that both InfluencingParameters and InfluencedVariableReferences are attached to the surrounding ServiceBehaviorAbstraction. InfluencingParameter is modeled as a subtype of element ModelVariable.

Relationships: Dependency and Dependency Propagation

As shown in Figure 4.26, we distinguish the two types of relationships DependencyRelationship and DependencyPropagationRelationship between model variables. The former

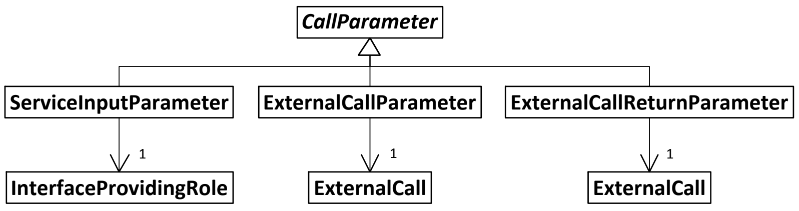


Figure 4.24: Call Parameter Hierarchy

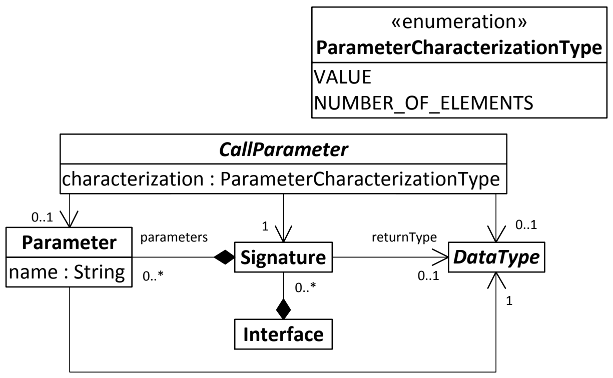


Figure 4.25: Call Parameters

declares one influenced variable to be dependent on one or more influencing parameters (the *independent* parameters). The latter connects one or more influencing parameters (the *independent* parameters) with one influencing parameter (the *dependent* parameter)) declaring the existence of a dependency between them. Thus, an influencing parameter can play the role of a dependent parameter in one relationship, while at the same time being an independent parameter in another relationship. Both DependencyRelationship and DependencyPropagationRelationship are non-symmetric, one-directional, transitive relationships. However, note that cycles of relationships are prohibited.

A Relationship is attached to the innermost InterfaceProvidingRequiringEntity, i.e., (composite) component or (sub-)system, that surrounds the relationship. A dependency is defined at the InterfaceProvidingRequiringEntity where the influenced variable resides, and is specified by the corresponding developer. A dependency propagation is specified for a ComposedStructure that is composed of several assembly contexts. Thus, both sides of the dependency propagation need to be identified not only by the InfluencingParameters but also by ComponentInstanceReferences indicating the specific component instances where the influencing parameters reside. As depicted in Figure 4.8 in Section 4.1.1, we require

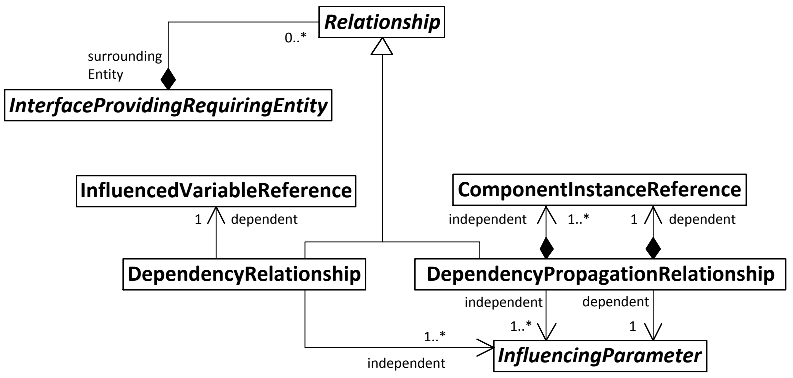


Figure 4.26: Relationships between Influenced Variables and Influencing Parameters

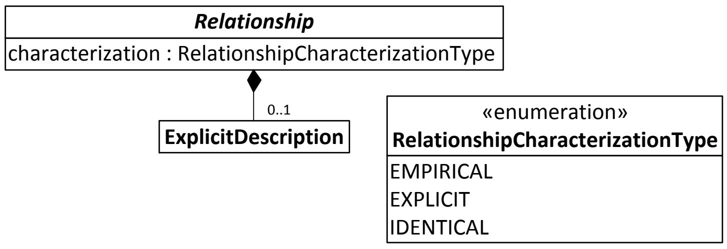

Relationship
characterization : RelationshipCharacterizationType
0..1
ExplicitDescription
«enumeration»
RelationshipCharacterizationType
EMPIRICAL
EXPLICIT
IDENTICAL

Figure 4.27: Characterization of Relationships

the specification of a *path* of assembly contexts in order to unambiguously identify a certain component instance from the perspective of a ComposedStructure.

Characterization of Relationships

A dependency or dependency propagation relationship is characterized via the model attribute RelationshipCharacterizationType. The relationship represents a function that maps the values of the relationship's independent variable(s) to a characterization of the relationship's dependent variable. In other words, a relationship with n independent variables $mv_1, \ldots, mv_n$ and one dependent variable d is characterized by a function that maps the n values of the independent variables to a random variable representing the dependent variable d. We distinguish three types of characterizations: (i) EMPIRICAL, (ii) EXPLICIT, and (iii) IDENTICAL.

EMPIRICAL means that the relationship is characterized using monitoring statistics, i.e., the functional relation is determined using empirical data. The function is accessed via an interface to the monitoring infrastructure. The interface is described in Section 4.1.6. In this chapter, we focus on the modeling concepts at the meta-model level. For the description of how the empirical characterizations can be derived from monitoring data, i.e., how the mentioned interface to the monitoring infrastructure can be implemented, we refer to Section 6.4.5.

EXPLICIT means that the relationship is characterized explicitly, i.e., with an explicit function specified using PCM's StoEx language. Such a characterization is suitable if an expression of the functional relation between the involved variables is available, the expression is modeled using model entity ExplicitDescription that corresponds to the PCM model element Expression (cf. Koziolek (2008, p.80)). Note that Scopes of involved ModelVariables are not considered when evaluating EXPLICIT relationships.

IDENTICAL is a special case of EXPLICIT in order to simplify modeling common delegations. It means that the independent variable of the relationship directly maps to the dependent variable of the relationship. This characterization type is thus only allowed if the relationship has exactly *one* independent variable.

Note that a model variable that is characterized as EXPLICIT can only be the dependent variable in relationships that are characterized as EXPLICIT, too. The model variable's explicit description is then overwritten by the explicit description of the relationship. Furthermore, if a model variable is the dependent variable in more than one relationship, the following constraints hold. At most one of these relationships is allowed to be characterized with type EXPLICIT. Otherwise, the explicit description would be ambiguous. If there is such an EXPLICIT relationship, relationships with characterization type EMPIRICAL are ignored. Thus, relationships whose characterization type is modeled as EXPLICIT take precedence over relationships whose characterization type is modeled as EMPIRICAL.

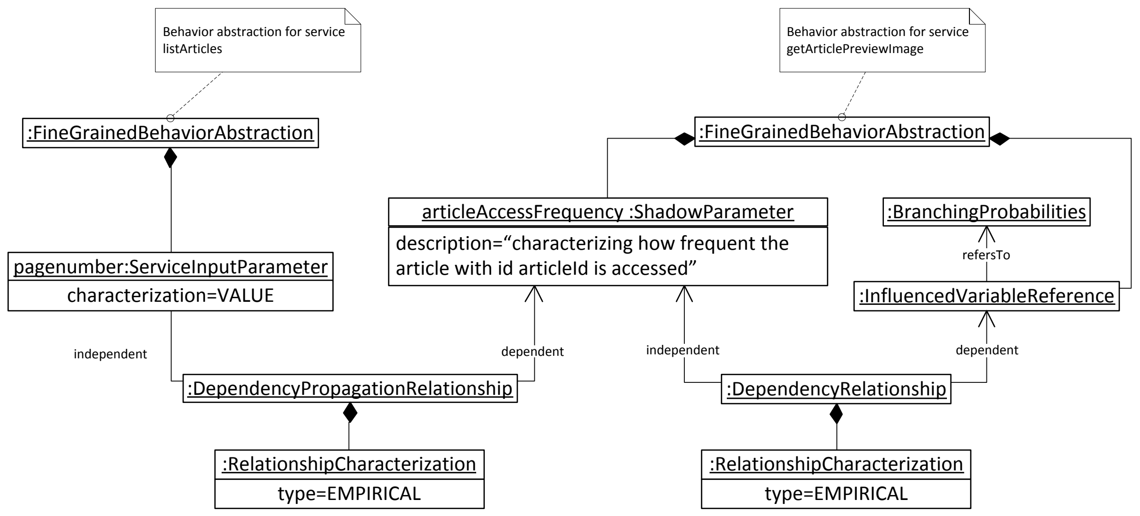

Behavior abstraction for service listArticles
Behavior abstraction for service getArticlePreviewImage
:FineGrainedBehaviorAbstraction
:FineGrainedBehaviorAbstraction
articleAccessFrequency :ShadowParameter
description="characterizing how frequent the article with id articleId is accessed"
:BranchingProbabilities
refersTo
:InfluencedVariableReference
pagenumber:ServiceInputParameter
characterization=VALUE
independent
dependent
independent
dependent
:DependencyPropagationRelationship
:DependencyRelationship
:RelationshipCharacterization
type=EMPIRICAL
:RelationshipCharacterization
type=EMPIRICAL

Figure 4.28: Example: Modeling Parameter Dependencies

4.1.5.4 Example

As example for parameter dependencies, we use the parameter dependencies as they are illustrated in Figure 4.22. The service behavior abstraction of service *getArticlePreviewImage* provided by the *JPAProvider* component contains an influenced variable. The influenced variable refers to the branching probabilities of the branch reflecting whether the requested data is in the cache or needs to be queried from the database (see Figure 4.20 for the branching behavior).

An excerpt of the object diagram is shown in Figure 4.28. There is a shadow parameter named *articleAccessFrequency*, also attached to the service behavior abstraction of service *getArticlePreviewImage*. The shadow parameter and the influenced variable are linked using a DependencyRelationship. The type of the RelationshipCharacterization is set to EMPIRICAL. The parameter *pagenumber* of service *listArticles*, provided by component *CatalogServlet*, is exposed as another InfluencingParameter. Then there is a DependencyPropagationRelationship with parameter *pagenumber* as independent parameter and the *articleAccessFrequency* parameter as dependent parameter. This DependencyPropagationRelationship is also characterized with type EMPIRICAL.

ShadowParameter *articleAccessFrequency* cannot be directly monitored. Thus, the DependencyPropagationRelationship and the DependencyRelationship cannot be characterized separately. The two relationships rather indicate that there is a dependency between service input parameter *pagenumber* and the branching probabilities of cache hit or miss. In Section 5.2, we describe how we resolve such relationships automatically.

Figure 4.29 shows exemplary monitoring statistics showing how values of parameter *pagenumber* ranging from 1 to 12 relate to the probability of a cache miss. For instance, for a *pagenumber* of 4, the probability of a cache miss is monitored to be 0.23 on average. Thus, for a *pagenumber* of 4, the BranchingProbabilities can be parameterized with `EnumPMF[('Branch1';0.77)('Branch2';0.23)]` as PMF. For a *pagenumber* of 8, the probability of a cache miss is monitored to be 0.9 on average, and the BranchingProbabilities can be parameterized with `EnumPMF[('Branch1';0.1)('Branch2';0.9)]` as PMF. For a description of how the empirical characterizations can be derived from monitoring statistics, see Section 6.4.5.

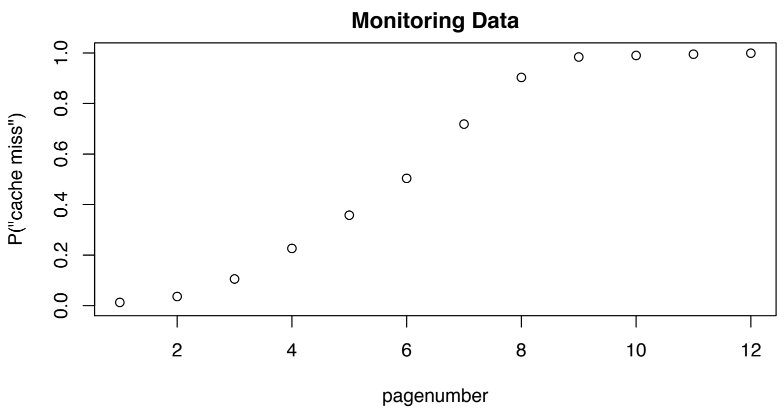


Figure 4.29: Example: Characterizing Parameter Dependencies

```
interface IApplicationLevelMonitor {
 RandomVariable getCharacterizationForModelVariable(
  ModelVariable modelVariable,
  ComponentInstanceReference compInstanceContainingMV);

 RandomVariable getCharacterizationForParameterDependency(
  List<ModelVariable> independentMVs,
  List<ComponentInstanceReference> compInstsContainingIndependentMVs,
  List<Literal> independentValues,
  ModelVariable dependentMV,
  ComponentInstanceReference compInstanceContainingDependentMV);
}
```

Listing 4.1: Monitoring Interface for the Application Level Model

4.1.6 Interface to Monitoring Infrastructure

The modeling abstractions for the application architecture level have ModelVariables and Relationships that need to be parameterized. In case the MVCharacterizationType of a ModelVariable, or the RelationshipCharacterizationType of a Relationship is set to EMPIRICAL, the characterization of the corresponding ModelVariable or Relationship is not specified in the application architecture model instance. Characterization type EMPIRICAL means that the values need to be obtained from monitoring statistics. Thus, the monitoring infrastructure of the running system has to be capable of providing appropriate monitoring statistics to characterize such ModelVariables and Relationships.

This section defines an interface named *IApplicationLevelMonitor* that needs to be implemented by the monitoring infrastructure in order to parameterize an application architecture model instance. Listing 4.1 shows the monitoring interface in Java syntax. The interface has two method signatures explained in the following.

The first method *getCharacterizationForModelVariable* returns the characterization for a given ModelVariable *mv* and a given component instance where *mv* resides. The characterization is returned as a RandomVariable. For example, consider the BranchingProbabilities of the BranchAction that is part of the fine-grained behavior depicted in Figure 4.15. The behavior describes service *calculateTotalCost* that is provided by component *ShoppingCartServlet*. For one component instance of *ShoppingCartServlet*, method *getCharacterizationForModelVariable* might return `EnumPMF[('Branch1';0.8)('Branch2';0.2)]`, for a different component instance of *ShoppingCartServlet*, the method might return `EnumPMF[('Branch1';0.6)('Branch2';0.4)]`. The implementation of the method needs to consider the ScopeSet of the given ModelVariable when returning the characterization as a RandomVariable. Note that a characterization of a model variable mv is assumed to be available via the method *getCharacterizationForModelVariable* if mv is a resource demand, a response time or a control flow variable, characterized as EMPIRICAL. If the characterization is not available for such model variables, a performance prediction cannot be conducted. If mv is an InfluencingParameter, however, the method may return $NULL$ and a performance prediction can still be conducted.

The second method *getCharacterizationForParameterDependency* returns the characterization for a given list of independent ModelVariables and one dependent ModelVariable. The method is used to obtain a characterization of a parameter dependency, returned as RandomVariable. A parameter dependency is determined by the method's arguments:

- a list of references to ModelVariables $mv_1, \ldots, mv_n$ that are the independent variables,
- a list of Literals $v_1, \ldots, v_n$ that are values for the above-mentioned variables,
- a list of ComponentInstanceReferences indicating where $mv_1, \ldots, mv_n$ reside,
- a reference to a ModelVariable d that is the dependent variable,
- a reference to a component instance indicating in which component instance d resides.

Formally, the method's arguments specify the signature of a n-ary function to be evaluated at arguments $v_1, \ldots, v_n$. The result of the function, returned as a random variable, characterizes the dependent variable. The implementation of the method needs to consider the ScopeSets of the involved ModelVariables when deriving a characterization for the dependent variable. If $n = 0$, then *getCharacterizationForParameterDependency* has the same semantics as *getCharacterizationForModelVariable*.

While this section described the methods of interface *IApplicationLevelMonitor*, Section 6.4 presents how the interface can be implemented.

4.2 Resource Landscape Model and Deployment Model

Having modeled the performance-relevant behavior of the software architecture as described in Section 4.1, predicting the system performance requires information about the execution environment (the resource landscape) where the software system is deployed.

Today's IT execution environments implement abstraction layers based on virtualization and middleware technologies. This promises benefits when reacting on changes of service workloads by provisioning and allocating resources as they are needed. However, introducing new abstraction layers to increase flexibility comes at the cost of increased system complexity. Existing model-based resource management and capacity planning techniques such as Becker et al. (2009); Kounev (2006) are targeted at design-time Quality of Service (QoS) analysis. In general, such approaches are not applicable at run-time because they abstract the complex infrastructure architecture or do not consider the dynamic aspects important for run-time system adaptation and resource management. For instance, the virtualization overhead (Huber et al., 2011b) that affects resource utilization and service response times is typically ignored.

Current techniques do not provide means to model the layers of the component execution environment (e.g., the virtualization layer) explicitly. The performance influences of the individual layers, the dependencies among them, and the resource allocations at each layer should be captured as part of the models. This is necessary in order to be able to

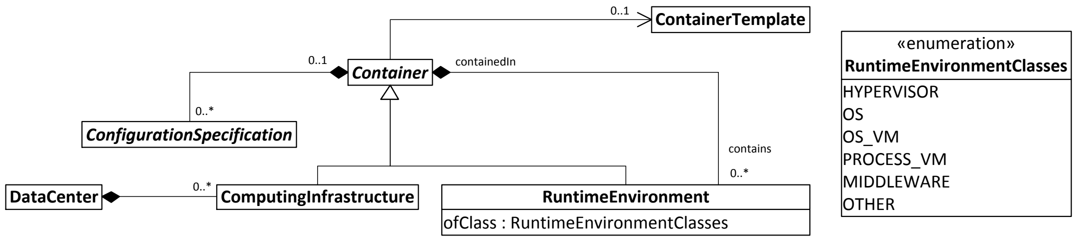


Figure 4.30: Hierarchical Run-time Environments in the Resource Landscape, cf. Huber (2014)

predict at run-time how a change in the execution environment (e.g., modifying resource allocations at the virtual machine level) would affect the system performance. The resource landscape meta-model, part of DML, allows modeling the physical and logical resources (e.g., virtualization and middleware layers) as they are provided by modern data centers.

Based on Huber et al. (2012a), we briefly describe the modeling abstractions in the following subsection. In Section 4.2.2, we provide an exemplary resource landscape where the *WebShop* of the running example presented in Section 4.1.2 is deployed.

4.2.1 Modeling Abstractions

The resource landscape meta-model reflects the static view of a distributed execution environment, i.e., it provides means to describe i) the computing infrastructure and its physical resources, and ii) the different layers within the system that provide logical resources, e.g., virtual machines, virtual CPUs, and so on.

Here, we show the core of the DML resource landscape meta-model. A detailed description is provided in Kounev et al. (2014). Figure 4.30 depicts the part of the meta-model describing the nested containment principle to model resource landscapes. For example, imagine servers that contain a virtualization platform and Virtual Machines (VMs), again containing an operating system, containing a middleware layer, and so on. This leads to a tree of nested entities that can change at run-time (e.g., when a VM is migrated). The central entity of this meta-model is the abstract entity Container. It is distinguished between two major types of containers, the ComputingInfrastructure and the RuntimeEnvironment. A ComputingInfrastructure forms the root element in our tree-like structure of containers and corresponds to a physical machine. It cannot be contained in another container but it can have nested containers. The RuntimeEnvironment is a generic model element to build nested layers, i.e., it can be contained within a container and it might itself contain further run-time environments. Each RuntimeEnvironment has a property to specify the class of the RuntimeEnvironment. The possible classes are listed in the enumeration type RuntimeEnvironmentClasses. The classes are *HYPERVISOR* for the different hypervisors of virtualization platforms, *OS* for operating systems, *OS_VM* for virtual machines emulating standard hardware, *PROCESS_VM* for virtual machines like the Java VM, *MIDDLEWARE* for middleware environments, and *OTHER* for any other type. The consistency within the modeled layers is ensured with Object Constraint Language (OCL) expressions.

Furthermore, the type Container has a property to specify its configuration. We distinguish three different types of configuration specifications: ActiveResourceSpecification, PassiveResourceSpecification, and CustomConfigurationSpecification (see Figure 4.31 (a)). The purpose of the ActiveResourceSpecification is to specify the active resources a Container provides. Using the properties schedulingPolicy, processingRate and numberOfParallelProcessingUnits, for example, a dualcore CPU can be specified with *PROCESSOR_SHARING* as schedulingPolicy, a processingRate of 2.4 GHz and a numberOfParallelProcessingUnits

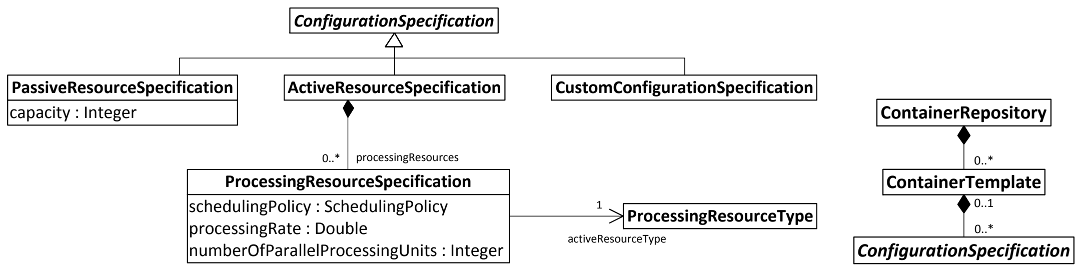

ConfigurationSpecification
PassiveResourceSpecification
capacity : Integer
ActiveResourceSpecification
CustomConfigurationSpecification
0..*
processingResources
ProcessingResourceSpecification
schedulingPolicy : SchedulingPolicy
processingRate : Double
numberOfParallelProcessingUnits : Integer
1
activeResourceType
ProcessingResourceType
ContainerRepository
0..*
ContainerTemplate
0..1
0..*
ConfigurationSpecification

Figure 4.31: Configuration Specification and Container Template Repository, cf. Huber (2014)

of 2. The PassiveResourceSpecification can be used to specify properties of passive resources. Passive resources can be, e.g., the main memory size or software resources such as database connection thread pools. The attribute capacity is used to specify, e.g., the number of threads. In case a Container has an individual configuration that cannot be modeled with the previously introduced elements, one can use the CustomConfigurationSpecification. The semantics of this entity is described in Huber et al. (2012a).

With the meta-model concepts presented so far, it would be necessary to model each container and its configuration explicitly. This can be very cumbersome, especially when modeling clusters of several identical machines. The template concept depicted in Figure 4.31 (b) enables the differentiation between container types and instances of these types. A container type specifies the general performance properties relevant for all instances of this type and the instances store the performance properties of each container instance. The template concept is implemented using the ContainerTemplate entity. Configured container templates are collected in the ContainerRepository. The ContainerTemplate is similar to a Container since it also may refer to ConfigurationSpecification. A Container in a resource landscape model instance might have a reference to a ContainerTemplate (see Figure 4.30). The advantage of this template concept is that the general properties relevant for all instances of one container type can be stored in the container template and the relevant configuration specific for an individual container instance can differ. For example, assume that a container instance has no individual properties and only a reference to a template. Then, only the configuration specification of the template would be considered. However, if the container instance has an individual configuration specification, then these settings would override the properties of the template. The template mechanism is useful in situations where, e.g., a VM is cloned (use template as configuration) and later changed (individual configuration options possible).

After describing the execution environment, one must specify which services are executed on which part of the resource landscape. We refer to this specification as *deployment* captured in the deployment model. The deployment model, based on PCM's allocation model (Becker et al., 2009), is shown in Figure 4.32. A Deployment connects a model instance of the software architecture (System) with a model instance of a resource landscape (DataCenter). It consists of several deployment contexts that map an AssemblyContext contained in the system model to a Container that is a model element of the resource landscape.

4.2.2 Example

Figure 4.33 shows an exemplary deployment of the running example introduced in Section 4.1.2 to a resource landscape model instance.

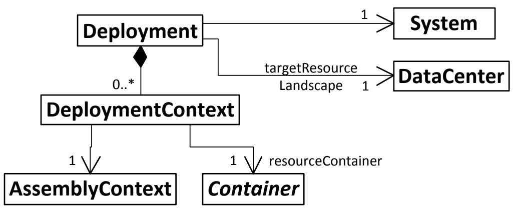


Figure 4.32: Deployment Model

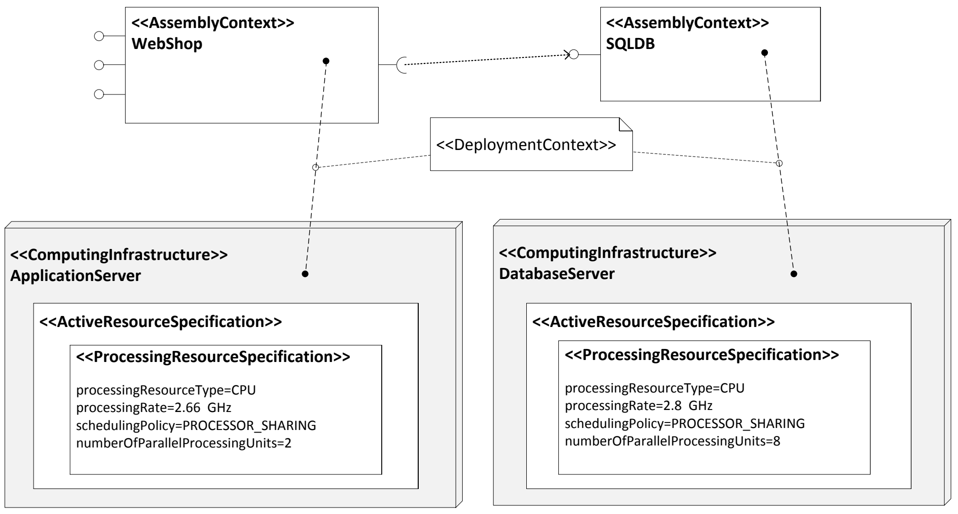

<<AssemblyContext>> WebShop
<<AssemblyContext>> SQLDB
<<DeploymentContext>>
<<ComputingInfrastructure>> ApplicationServer
<<ActiveResourceSpecification>>
<<ProcessingResourceSpecification>>
processingResourceType=CPU
processingRate=2.66 GHz
schedulingPolicy=PROCESSOR_SHARING
numberOfParallelProcessingUnits=2
<<ComputingInfrastructure>> DatabaseServer
<<ActiveResourceSpecification>>
<<ProcessingResourceSpecification>>
processingResourceType=CPU
processingRate=2.8 GHz
schedulingPolicy=PROCESSOR_SHARING
numberOfParallelProcessingUnits=8

Figure 4.33: Example: WebShop Deployment

There is a datacenter that consists of two ComputingInfrastructures, namely an *Application-Server* and a *DatabaseServer*. The servers' CPUs are modeled as active resources. The CPU of the *ApplicationServer* consists of two processing units at a processing rate of 2.66 GHz. The *DatabaseServer* CPU has eight processing units at a processing rate of 2.8 GHz. The *WebShop* instance is deployed on the *ApplicationServer* node, a *SQLDB* instance is deployed on to the *DatabaseServer* node.

4.3 Usage Profile Model

Using the application architecture meta-model (Section 4.1) and the resource landscape and deployment meta-models (Section 4.2), an architecture-level performance model can be described. In order to conduct a performance prediction for a certain workload, the workload needs to be specified. This is done using the usage profile meta-model, also part of DML. The usage profile meta-model is based on PCM's usage model (Becker et al., 2009) and allows modeling user interactions with the system.

4.3.1 Modeling Abstractions

The modeling abstractions are shown in Figure 4.34. A UsageProfileModel consists of one or more UsageScenarios and references a System. A UsageScenario refers to a description of a WorkloadType (either an open workload or a closed workload (Schroeder et al., 2006)) and a ScenarioBehavior.

An open workload is characterized with an inter-arrival time specified as RandomVariable. A closed workload is characterized with a client population and a client think time also specified as RandomVariable.

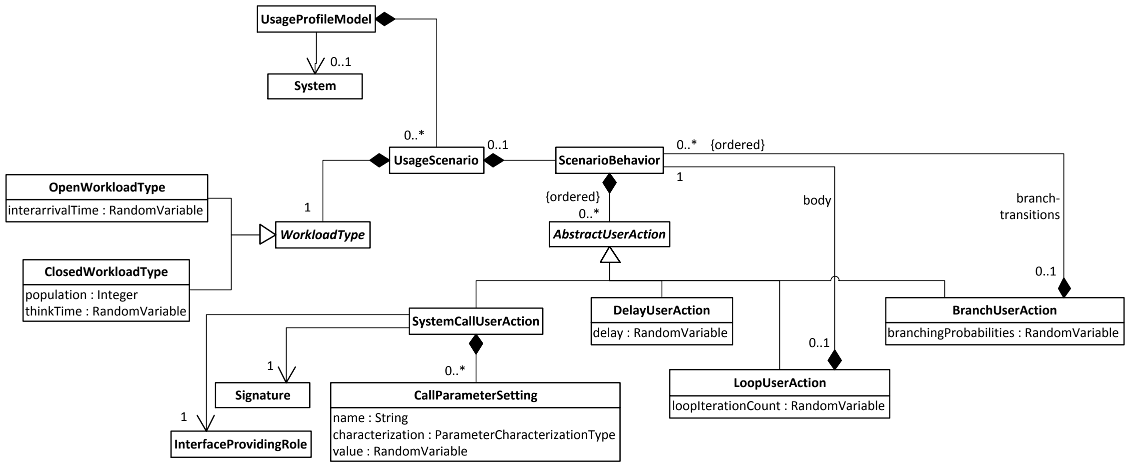


Figure 4.34: Usage Profile Model, cf. Becker et al. (2009)

A ScenarioBehavior allows describing which services of the referenced System are called by the user using so-called SystemCallUserActions. Such a SystemCallUserAction refers to a service signature of an interface that is provided by the system. The input parameters of the service signature can (but do not have to) be set using CallParameterSettings. A CallParameterSetting consists of the name of the parameter, a parameter characterization type (see Figure 4.25) and the value of the parameter as a random variable. The SystemCallUserActions can be arranged in a control flow with delays, branches and loops. Delays, as well as branching probabilities and loop iteration counts are described with a RandomVariable.

Note that a RandomVariable is an atom of a stochastic expression (see Section 4.1.4.3). It allows characterizing discrete probability distributions with PMFs, approximating continuous probability densities with samples, or parameterizing common PDFs such as the exponential distribution or binomial distribution.

4.3.2 Example

An example of a usage profile model is illustrated in Figure 4.35. It is a usage profile for the *WebShop* system presented in Section 4.1.2. It consists of one UsageScenario with an open workload. The users have an inter-arrival time of `Exp(1.0)`, i.e., an exponentially distributed inter-arrival time with mean 1.0. The user behavior starts with a call of the *listArticles* service, followed by a delay of `1000` and a service call to *viewArticleDetails*. The value of the *pagenumber* parameter of the *listArticles* service is parameterized using a PMF. With a probability of 0.5 each, either the first page or the second page is requested.

4.4 Summary

In this chapter, we described architecture-level performance abstractions for use in online scenarios. The proposed performance abstractions reflect the application architecture, its deployment to a resource landscape, as well as the application's usage profile, thus making it possible to predict the impact of workload changes, service (re-)compositions, software configuration changes, or changing resource allocations.

In Section 4.1, we presented modeling abstractions for the performance-relevant factors of a component-based application architecture (Brosig et al., 2013b, 2012), specifically for use at run-time. This includes a new approach to model performance-relevant service behavior at different levels of granularity (see Section 4.1.3), a new approach to parameterize performance-relevant properties of software components (see Section 4.1.4) and a new

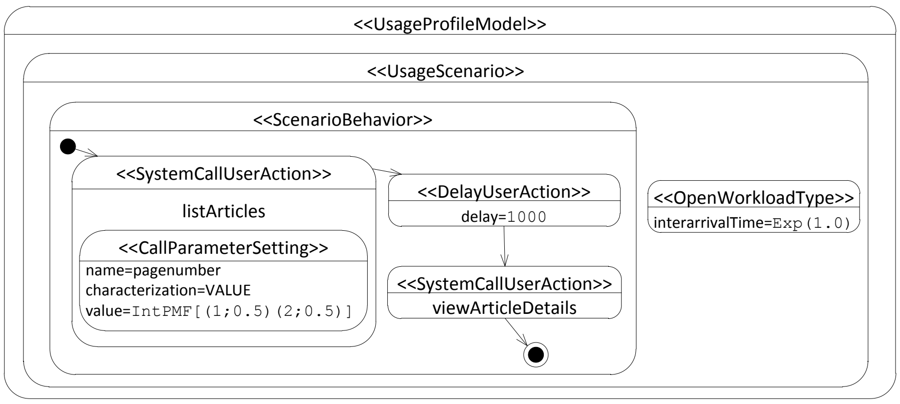


Figure 4.35: Example: Usage Profile Model Instance

approach to model dependencies between parameters (see Section 4.1.5). The presented modeling abstractions are part of the Descartes Modeling Language (Kounev et al., 2014), a new modeling language for run-time performance and resource management of modern dynamic IT service infrastructures.

Parts of DML that allow describing the resource landscape and the deployment are briefly presented in Section 4.2. Modeling abstractions for the description of the system workload, i.e., for the description how the system is used and with which intensity it is used, are shown in Section 4.3.

5. Online Prediction Techniques

To ensure that a software system meets its performance requirements, the ability to predict its performance under different configurations and workloads is highly valuable throughout the system life cycle from the design phase to system operation.

At software design time, a software architect searches for a suitable assembly of components to build a software system. Using an architecture-level performance model of the system, the software architect can simulate different assemblies and configurations to predict the performance behavior. The predictions are not constrained to complete within strict time bounds, but should allow qualitative comparisons of design alternatives with high accuracy (Becker et al., 2009; Reussner et al., 2011). Furthermore, the software architect may optimize compositions or configuration parameter settings on the model level. At deployment time, a software deployer sizes the resource environment so that the system on the one hand satisfies performance objectives and on the other hand does not waste resources. Using a performance model, this system sizing and capacity planning step can be facilitated. Expensive performance tests can be avoided, because different load situations can be simulated on different resource settings.

A proactive online performance and resource management, however, aims at adapting system configuration and resource allocations dynamically. Overload situations should be anticipated, suitable reconfigurations should be found on the model level and triggered *before* Service Level Agreements (SLAs) are violated (Thereska et al., 2005; Kounev et al., 2010). *Online performance predictions* need to be conducted to answer questions such as: What performance would a new service or application deployed on the infrastructure exhibit and how much resources should be allocated to it? How should the workloads of the new service/application and existing services be partitioned among the available resources so that performance requirements are satisfied and resources are utilized efficiently? What would be the performance impact of adding a new component or upgrading an existing component as services and applications evolve? If an application experiences a load spike or a change of its workload profile, how would this affect the system performance? Which parts of the system architecture would require additional resources? What would be the effect of migrating a service or an application component from one physical server to another? However, there is a trade-off between prediction accuracy and time-to-result. There are situations where the prediction results need to be available very fast to adapt the system *before* SLAs are violated. An accurate fine-grained performance prediction comes at the cost of higher prediction overhead and longer prediction durations. More coarse-grained performance predictions can speed up the prediction process. Thus, in online scenarios, considering this trade-off when conducting performance prediction is desirable.

In this chapter, we describe how to conduct performance predictions *online* using the performance modeling abstractions described in the previous Chapter 4. Figure 5.1 provides a high-level overview of the prediction process showing the individual steps and their inputs and outputs.

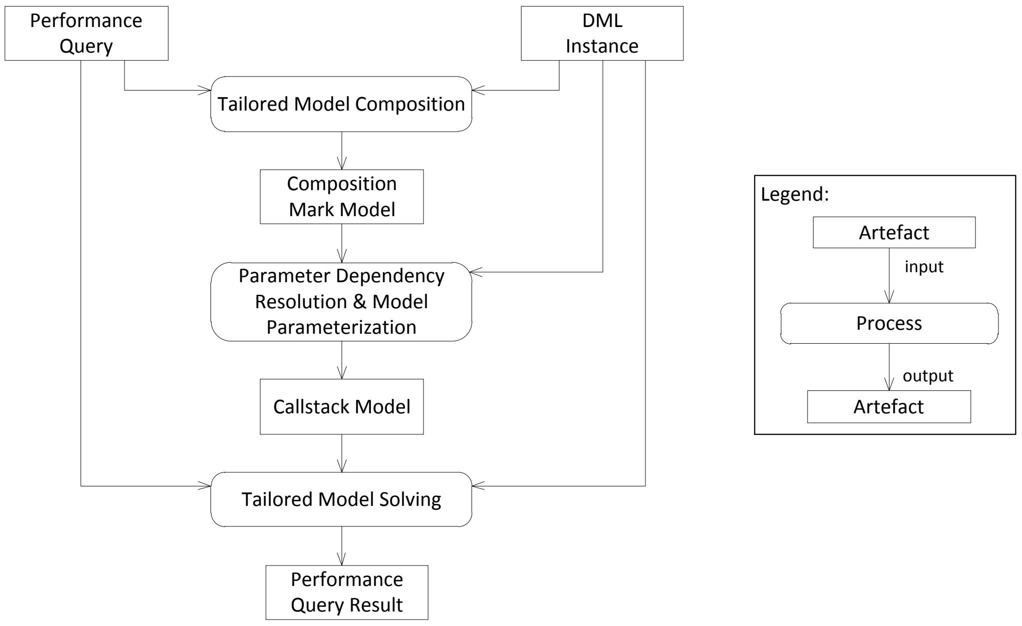


Figure 5.1: Online Performance Prediction Process

We assume an architecture-level performance model including a usage profile model as described in Chapter 4 to be available. The prediction process is triggered by a *performance query*. A performance query specifies which performance metrics should be predicted, e.g., which service response times need to be predicted, if response time percentiles are requested or average response times are sufficient, or for which resources the utilization should be predicted. TheFurthermore, a performance query may include a specification of how to trade-off between prediction accuracy and time-to-result, indicating if the query result needs to be available very fast at the expense of prediction accuracy or a longer prediction process with higher overhead is acceptable.

The prediction process starts with a model composition step. The step marks those parts of the performance model relevant for answering the query. These markings are kept in a composition mark model which serves as input for the next step. For instance, if a service is described with multiple service behaviors such as a fine-grained behavior, a coarse-grained behavior, and a black-box behavior, the model composition step chooses a description that provides adequate means to predict the requested performance metrics. Since the model composition also takes the trade-off specification of the given performance query into account, we refer to it as *tailored* model composition.

Having determined those parts of the architecture-level performance model that have to be considered for a performance prediction, the next step traverses the model starting with the usage scenarios specified as part of the usage profile model. On the one hand, it resolves involved parameter dependencies. On the other hand, it derives empirical characterizations for model variables via the interface described in Section 4.1.6. The output is a call graph with each involved model variable characterized by a random variable, represented as callstack model. The callstack model furthermore determines how the performance model has to be traversed for the performance prediction.

The next step is the tailored model solving step, i.e., it predicts the requested metrics considering the given trade-off specification. It uses existing model solving techniques based on established stochastic modeling formalisms. The model solving decides which concrete model solving technique to apply. In addition, model solving techniques also come with their own configuration options and can also be tailored to the performance query. Therefore, for each model solving technique and its configuration options, it is important to understand how it affects the performance prediction in terms of specific predictable metrics, prediction accuracy, and prediction overhead.

- A fine-grained simulation can provide the best prediction accuracy but has the lowest prediction speed compared to other solving techniques. Complex performance metrics such as response time distributions can be provided. A simulation can be accelerated by reducing the length of the simulation and/or the amount of collected simulation log data to the minimum that is required to predict the requested metrics considering the desired prediction accuracy. For instance, in case only mean value metrics are requested, the simulation can abstract from complex control flow constructs such as branches or loops.

- Analytical model solvers typically have lower prediction overhead compared to simulation, but they are often restricted in terms of the predictable metrics and the assumptions they make about the model input parameters. Analytical solvers such as LQNS (Franks et al., 1996) often assume exponentially distributed service times and request inter-arrival times (Balsamo et al., 2004), or have limited capabilities to analyze complex behavior such as blocking or simultaneous resource possession (Menasce and Virgilio, 2000; Li et al., 2009; Gilmore et al., 2005).

As analytical solving techniques, we apply asymptotic bounds analysis (Bolch et al., 1998), and make use of the analytical solver tool LQNS (Franks et al., 2011, 1996, 2009). A bounds analysis can quickly provide asymptotic bounds for the average throughput and the average response time, but this comes at the cost of lower accuracy. However, the results can still be accurate enough to make quick decisions when approximate performance results are sufficient (Bolch et al., 1998; Menasce and Virgilio, 2000). The LQNS solver implements several analytical solving techniques such as Mean Value Analysis (MVA) (Bolch et al., 1998) and combines the advantages of other existing analytical solvers, namely SRVN (Woodside et al., 1995) and MOL (Rolia and Sevcik, 1995). Given that LQNS is a solver for Layered Queueing Networks (LQNs), a transformation to LQNs has to be provided. As simulation technique, we transform the performance model to a Queueing Petri Net (QPN) (Kounev, 2006) and simulate it using the SimQPN simulation engine (Spinner et al., 2012). Both the transformation itself and the simulation are tailored to the given performance query.

The results of the tailored model solving step, i.e., the predicted performance metrics, are then returned to the query issuer.

The remainder of this chapter is organized as follows: Section 5.1 presents the tasks of the model composition step. Section 5.2 describes how the parameter dependencies and empirical characterizations introduced in Section 4.1.5 are handled. Section 5.3 formally presents model solving techniques. The tailoring mechanisms balancing the trade-off between prediction accuracy and prediction overhead, both for the model composition step as well as for the model solving step, are described in Section 5.4. Section 5.5 formalizes the notion of a performance query and introduces the Descartes Query Language (DQL), a language to express requested performance metrics as well as the goals and constraints in a specific prediction scenario.

5.1 Model Composition

The usage profile model described in Section 4.3 consists of one or more ScenarioBehaviors. A ScenarioBehavior describes which system services are called. A system service call translates to a service behavior abstraction that itself may refer to further service behavior abstractions via service calls denoted as external calls. Thus, the usage profile's ScenarioBehaviors determine those parts of the architecture-level performance model that have to be considered in a given performance prediction scenario.

However, using the model abstractions described in Section 4.1.3), for one service behavior there can be up to three different behavior descriptions of different granularity. A service behavior may be described by the three abstraction levels FineGrainedBehavior, CoarseGrainedBehavior, and BlackBoxBehavior. This ambiguity is resolved by the tailored model composition step. The decision which service behavior should be selected depends on the performance query, more specifically, on the specified trade-off between prediction accuracy and prediction overhead. The selected service behaviors are then *marked* in a so-called composition mark model. Figure 5.2 summarizes the input and the output of the model composition step.

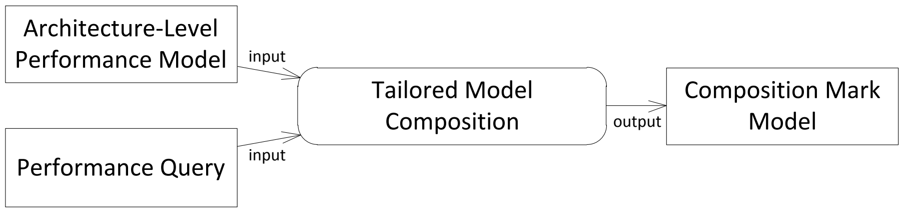

Architecture-Level Performance Model
input
Performance Query
input
Tailored Model Composition
output
Composition Mark Model

Figure 5.2: Input and Output of Model Composition

This section only presents the tasks of the model composition step. The tailoring mechanism itself is described later in Section 5.4.

5.2 Parameter Dependency Resolution and Model Parameterization

The Relationships as presented in Section 4.1.5 describe the performance-relevant behavior of a service's implementation depending on input parameters passed directly or indirectly upon service invocation. However, with common predictive performance models (see Section 2.2) such as Queueing Networks (QNs) it is not possible (i) to annotate individual requests with parameter settings, and (ii) to re-calculate influenced model parameters such as resource demands with, e.g., arithmetic expressions. Thus, Relationships cannot be directly translated to such modeling formalisms.

Since we want to apply model solving techniques based on established stochastic modeling formalisms, a pre-processing step needs to be introduced. The pre-processing step has two goals: (i) It resolves relationships between parameter dependencies. (ii) It derives empirical characterizations for unknown model variables via the interface described in Section 4.1.6. Starting with the usage profile model, parameter settings are propagated across relationships. As a result, the model variables are characterized with concrete probability distributions, obtained using empirical monitoring data, taking into account possible influencing parameters.

5.2.1 Input and Output

The input of the pre-processing step, as depicted in Figure 5.3, is an architecture-level performance model together with a mark model indicating which parts of the performance model need to be considered. The mark model stems from the preceding model composition step.

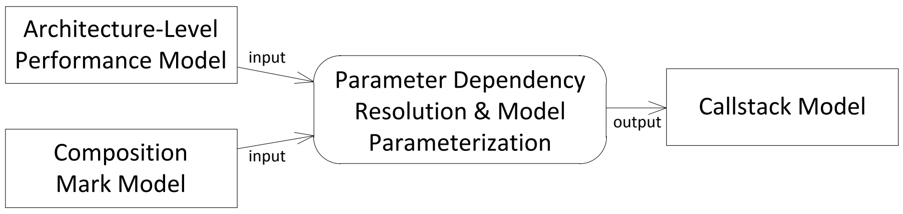

Architecture-Level Performance Model
input
Composition Mark Model
input
Parameter Dependency Resolution & Model Parameterization
output
Callstack Model

Figure 5.3: Input and Output of the Parameter Dependency Resolution and Model Parameterization Step

The output is modeled as a callstack model in form of a typical stack frame layout (Silberschatz et al., 2008). One stack frame represents one called ServiceBehaviorAbstraction. Values for the model variables of the ServiceBehaviorAbstraction are stored as part of the stack frame. The values are random variables characterized as concrete probability distributions taking into account possible influencing parameters. The purpose of the callstack model is twofold. On the one hand, it determines how the performance model has to be traversed for the performance prediction. On the other hand, it provides values for the involved model variables.

Figure 5.4 depicts the stack frame layout of the callstack model. Each called service provided by a component instance is represented by a StackFrame. A StackFrame provides information about the values of ModelVariables as ValueMapEntries. A ValueMapEntry assigns a RandomVariable to a ModelVariable. In case the ValueMapEntry is related to other ModelVariables, for each such ModelVariable there is a reference to a corresponding ValueMapEntry of that particular ModelVariable. Furthermore, a StackFrame refers to its predecessor StackFrame and to its successor StackFrames. For each ExternalCall, there is one successor StackFrame.

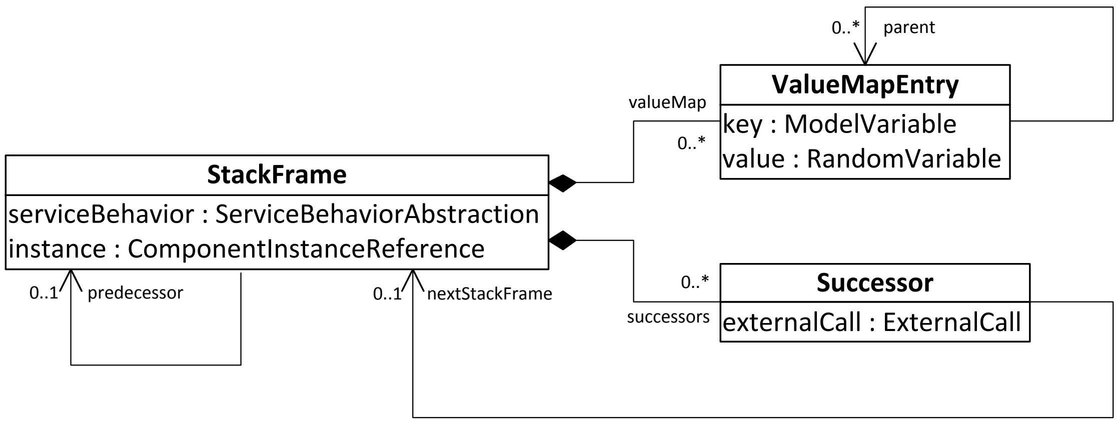


Figure 5.4: Stackframe Model

Koziolek (2008, p.134) describes a *DependencySolver* that solves the explicit parameter dependencies as they are used in Palladio Component Model (PCM). The DependencySolver propagates service input parameters that are passed from a requiring service to the provided service as well as the service output parameters that are passed from a provided service back to the requiring service. However, in our case, using the modeling abstractions described in Section 4.1, parameter dependency resolution involves parameter propagation across given Relationships and finding their transitive hull, as described in the next section. Furthermore, in our case, model variables as well as relationships can be empirically characterized, making it necessary to access monitoring data via the interface described in Section 4.1.6.

5.2.2 Relationship Resolution

This section provides a description how relationships are resolved and the model is parameterized. The architecture-level performance model is traversed starting with the usage profile model (see Section 4.3). For each SystemCallUserAction of a UsageScenario, the target service's behavior description is obtained. Given the markings of the composition mark model, the target service behavior is always unambiguous. Then, for a service behavior model, all containing ModelVariables are processed, i.e., values of the ModelVariables are determined taking Relationships into account. If a service behavior model contains calls to other services, i.e., ExternalCalls, their target services are traversed as well. While traversing the architecture-level performance model, the callstack output model is filled accordingly.

Before describing the process in detail, we use the running example of Chapter 4 to illustrate the inputs and outputs, tasks and challenges. The usage profile model instance is depicted in Figure 4.35, calling two services *listArticles* and *viewArticleDetails*. Assume the requested metrics of the given performance query are the average response times for the two services as well as the average CPU utilization of the application server and database server as they are shown in the deployment in Figure 4.33. Figure 4.22 shows parameter dependencies in the context of service *listArticles*, propagating parameter *pagenumber* of service *listArticles* to model variable BranchingProbabilities of the service behavior of service *getArticlePreviewImage* (see Figure 4.20 for the fine-grained behavior description of service *getArticlePreviewImage*). Model variable BranchingProbabilities indicates whether a database query is issued or not. The parameter dependency is modeled using a DependencyProgagationRelationship between parameter *pagenumber* and a ShadowParameter named *article access frequency*, and a *DependencyRelationship* between the ShadowParameter and model variable BranchingProbabilities.

In this example, the pre-processing step of parameter dependency resolution and model parameterization has to do the following:

- In order to make a performance prediction, values for involved ControlFlowVariables, ResourceDemands or ResponseTimes need to be available. Thus, such model variables need to be characterized for all involved service behaviors. Characterizing a model variable means obtaining a random variable via interface *IApplicationLevelMonitor* (see Section 4.1.6) if the variable's characterization type is modeled as EMPIRICAL, or calculating the random variable from the stochastic expression that is attached to the variable if the variable's characterization type is modeled as EXPLICIT.

  In the example, this includes the fine-grained behavior of service *listArticles* (see Figure 4.21), the fine-grained behavior of service *getArticlePreviewImage* (see Figure 4.20), as well as behavior descriptions for services *issueNamedQuery_previewImage* and *viewArticleDetails*. Note that for the latter two services, illustrations of their behavior descriptions are not provided.

- In case a model variable is modeled to be dependent on an influencing parameter via relationships, the parameter dependency should be considered, i.e., the random variable characterizing the model variable should reflect a conditional probability distribution. If the dependency cannot be resolved, as a fallback the model variable can still be characterized *without* considering the dependency. The method *getCharacterizationForParameterDependency* of interface *IApplicationLevelMonitor* is called to obtain a conditional probability distribution, while the method *getCharacterizationForModelVariable* is used in the fallback case.

In the example, the DependencyRelationship and the DependencyPropagationRelationship need to be transitively resolved to reveal the dependency between input parameter *pagenumber* of service *listArticles* and the BranchingProbabilities of service *getArticlePreviewImage*. In the usage profile model, the value of the *pagenumber* parameter of the *listArticles* service is parameterized using a Probability Mass Function (PMF). With a probability of 0.5 each, either the first page or the second page is requested. The characterization of the BranchingProbabilities should thus be a conditional probability distribution, taking the information about the *pagenumber* into account. As shown in Figure 4.29, for a small *pagenumber*, the probability of a cache access is much higher than for a large *pagenumber*.

Algorithm 1 to Algorithm 5 describe the process in pseudocode. Note that we use the expression $\mathit{propertyname}(\mathit{inst})$ to denote property *propertyname* of a type instance *inst*. In case the property has multiplicity > 1, $\mathit{propertyname}(\mathit{inst})$ evaluates to a set. For example, let f be an instance of type StackFrame, then expression $\mathit{valueMap}(f) \leftarrow \emptyset$ means that an empty set is assigned to property valueMap of f.

5.2.2.1 Model Traversal

First, we describe how the architecture-level performance model is traversed and stack frames of the callstack model are created.

We start with the traversal of a UsageScenario's user actions. Method *Solve* in Algorithm 1 is called whenever a SystemCallUserAction is reached. The method has a SystemCallUserAction as input parameter. For the system call, first the target service behavior is obtained, then the parameter settings are processed: Each CallParameterSetting of the SystemCallUserAction translates into a value map entry of the stack frame that is initialized for the target service. In other words, method *Solve* prepares a stack frame for the next service behavior and injects parameter settings as they are specified in the usage scenario. Afterwards, the model traversal proceeds with traversing the target service behavior by calling the method described in Algorithm 2.

```
1  $Solve(sc : \text{SystemCallUserAction})$
2  begin
3      $t \leftarrow$ target service behavior abstraction for $sc$
4      $f \leftarrow$ initialize new stack frame for $t$
5      foreach call parameter setting $s$ of $sc$ do
6          $p \leftarrow$ influencing parameter referenced by $sc$ and $s$
7          $e \leftarrow$ new value map entry $e$ with
8              $key(e) \leftarrow p$ and $value(e) \leftarrow value(s)$
9          $parent(e) \leftarrow \emptyset$
10         $valueMap(f) \leftarrow valueMap(f) \cup \{e\}$
11     end
12     $Solve(t, f)$
13     return $f$
14 end
```

Algorithm 1: Traversal Starting with SystemCallUserAction

Algorithm 2 shows method *Solve* with a ServiceBehaviorAbstraction and its corresponding stack frame as input. The ServiceBehaviorAbstraction is processed in two steps. First, the model variables are processed. Then, calls to external services (i.e., ExternalCalls) are followed. For each call to an external service, the target service behavior is obtained and its stack frame is initialized as successor. Note that the traversal is independent of the component-internal control flow of the ServiceBehaviorAbstraction. Even if an external call is modeled within a loop of a FineGrainedBehavior description, the external call is processed only once.

```
1  $Solve(sb : \text{ServiceBehaviorAbstraction}, f : \text{StackFrame})$
2  begin
3      $ProcessModelVariables(sb, f)$
4      foreach external call $e$ in $sb$ do
5          $t \leftarrow$ target service behavior abstraction for $e$
6          $f' \leftarrow$ initialize new stack frame for $t$
7          $predecessor(f') \leftarrow f$
8          $s \leftarrow$ new successor $s$ with
9              $externalCall(s) \leftarrow e$ and $nextStackFrame(s) \leftarrow f'$
10         $successor(f) \leftarrow successor(f) \cup \{s\}$
11         $Solve(t, f')$
12     end
13     return $f$
14 end
```

Algorithm 2: Traversal of ServiceBehaviorAbstractions

5.2.2.2 Processing Model Variables

Processing the model variables of a service behavior means obtaining values for the model variables and storing them in the corresponding stack frame. Relationships between model variables need to be resolved and empirical characterizations for the model variables need to be derived. As a result, the values of the model variables are characterized with concrete probability distributions taking into account possible influencing parameters.

The processing of model variables is described in Algorithm 3. Method *ProcessModel-Variables* has a service behavior and a corresponding stack frame as input. The method processes all model variables that are located in the given service behavior. For each model variable, the method obtains a value and creates a corresponding value map entry in the given stack frame, unless there is already a corresponding value map entry in the stack frame available.

For each InfluencingParameter p, the method searches for the set of dependency propagation relationships that refer to p as dependent parameter. This set is then processed by method *ProcessModelVariableWithRelationships*, using the model variable p, the set of referring relationships, the stack frame of the current service behavior, and a boolean flag *nullValuesAllowed*, as input. Setting the boolean flag to *TRUE* indicates that null values are allowed for p (see Algorithm 4). Note that an InfluencingParameter may have a null value when triggering a performance prediction which is in contrast to ControlFlowVariables, ResourceDemands or ResponseTimes (see Figure 4.18).

For each InfluencedVariableReference v, the method *ProcessModelVariables* searches for the set of dependency relationships that refer to v as dependent variable. The model variable v is referring to is then also processed by method *ProcessModelVariableWithRelationships*. This time, the boolean argument is set to *FALSE* indicating that null values are not allowed since v can only point to a ControlFlowVariable, ResourceDemand or ResponseTime (see Figure 4.23).

Having processed all model variables that are dependent variables of Relationships, the remaining model variables of the given service behavior are then processed. The value

```
ProcessModelVariables(sb : ServiceBehaviorAbstraction, f : StackFrame)
begin
    foreach influencing parameter p in sb do
        if value map entry e in f with
            key(e) = p and value(e) ≠ NULL already exists then
            break
        end
        R ← all dependency propagation relationships
              with p as dependent parameter
        ProcessModelVariableWithRelationships(p, f, R, TRUE)
    end
    foreach influenced variable reference v in sb do
        m ← referencedModelVariable(v)
        if value map entry e in f with
            key(e) = m and value(e) ≠ NULL already exists then
            break
        end
        R ← all dependency relationships with v as dependent variable
        ProcessModelVariableWithRelationships(m, f, R, FALSE)
    end
    foreach model variable m in sb not yet processed do
        if value map entry e in f with
            key(e) = m and value(e) ≠ NULL already exists then
            break
        end
        e ← new value map entry e with key(e) ← m
        if characterization(m) = EMPIRICAL then
            value(e) ← getCharacterizationForModelVariable(m)
        else
            value(e) ← description(explicitDescription(m))
        end
        parent(e) ← ∅
        valueMap(f) ← valueMap(f) ∪ {e}
    end
end
```

Algorithm 3: Processing the ModelVariables of a ServiceBehaviorAbstraction

for such a model variable is obtained using monitoring method *getCharacterizationForModelVariable* introduced in Section 4.1.6, if the model variable's characterization type is EMPIRICAL. If the characterization type is modeled as EXPLICIT, the value is read from the ExplicitDescription attached to the model variable. Note that the value of a model variable m is assumed to be always available via the monitoring interface if m is a resource demand, a response time or a control flow variable, characterized as EMPIRICAL. If m is an InfluencingParameter, the monitoring interface can return *NULL*, see Section 4.1.6 for more details.

As a result of method *ProcessModelVariables*, for each model variable contained in the given service behavior there is a value map entry in the given stack frame.

For the sake of simplicity, we omitted information about component instances in the pseudocode representations of the algorithms. However, as depicted in Figure 5.4, each StackFrame has a property instance that represents the current component instance.

Algorithm 4 describes method *ProcessModelVariableWithRelationships.* It has four input parameters: a model variable m, a stack frame, a set of relationships that refer to m as dependent variable, and a flag indicating if it is allowed to assign null values to model variable m or not. The method obtains a value for the given model variable m, taking the relationships R that may influence m into account. Relationships whose characterizations are modeled as EXPLICIT take precedence over relationships whose characterizations are modeled as EMPIRICAL (see Section 4.1.5). Furthermore, at most one relationship with characterization EXPLICIT is allowed for a model variable.

Starting with a relationship of EXPLICIT characterization if available, method *ProcessModelVariableWithRelationships* looks for values of the relationship's influencing parameters (denoted as independent parameters). This lookup is done in the value map entries of the given stack frame and its predecessors. If the lookup is successful, the resulting value of model variable m is calculated. The calculation of arithmetic operations on the random variables translates to convolutions of the probability distributions, for details we refer to Koziolek (2008, Section 3.3.6). Having calculated the result, a new value map entry e for m is created. Furthermore, property parent is set: each independent variable of the relationship is added to e as parent. Since only one relationship with EXPLICIT characterization is allowed, the method returns when a value map entry has been created.

In case there is no relationship of characterization EXPLICIT or no result could be calculated for such a relationship, the method processes relationships with EMPIRICAL characterization. First, it collects all independent parameters of all EMPIRICAL relationships. Then, it obtains value map entries for such parameters from the given stack frame and its predecessors. For this set of value map entries, method *ResolveNullValuesTransitively* is called. For value map entries with a null value, this method traces back existing dependency propagation relationships in order to search for independent parameters where non-null values are available. The method is described in Algorithm 5 and explained in detail later.

Using the result set T' of method *ResolveNullValuesTransitively*, method *getCharacterizationForParameterDependency* (see Section 4.1.6) is called to get a value for model variable m. We use the notation *getCharacterizationForParameterDependency*(T', m) to denote a call to *getCharacterizationForParameterDependency* where m is the dependent variable and value map entries T' represent the set of independent parameters together with their independent values. If this is not successful, the result is *NULL*. This means, that no relationship between the influencing parameters and m can be monitored. If the result is *NULL* and the method input parameter *nullValuesAllowed* is *FALSE*, the value for m is obtained using method *getCharacterizationForModelVariable*, i.e., *without* considering any influencing parameters. The value for m is then stored in a new value map entry

```
$ProcessModelVariableWithRelationships($
      $m$ : ModelVariable, $f$ : StackFrame,
      $R$ : Set(Relationship), $nullValuesAllowed$ : Boolean)
begin
    foreach $r \in R$ with $characterization(r) = EXPLICIT$ do
        $T \leftarrow$ value map entries of influencing parameters $independent(r)$
              in the stack frame of $f$ and its predecessors
        if $T \neq \emptyset$ then
            $res \leftarrow Calculate(r, T)$
            $e \leftarrow$ new value map entry $e$ with
                  $key(e) \leftarrow m$ and $value(e) \leftarrow res$
            $parent(e) \leftarrow T$
            $valueMap(f) \leftarrow valueMap(f) \cup \{e\}$
            return
        end
    end
    $R_{emp} \leftarrow \{ r \in R$ with $characterization(r) = EMPIRICAL \}$
    if $R_{emp} \neq \emptyset$ then
        $S \leftarrow \bigcup_{r \in R_{emp}} independent(r)$
        $T \leftarrow$ value map entries of influencing parameters $S$
              in the stack frame of $f$ and its predecessors
        $T' \leftarrow ResolveNullValuesTransitively(T)$
        $res \leftarrow getCharacterizationForParameterDependency(T', m)$
        // result may be NULL, meaning that no relationship can be
           found to characterize the model variable
        $V = \emptyset$
        if $nullValuesAllowed = FALSE$ and $res = NULL$ then
            // fallback: try characterization without any relationship
            $res \leftarrow getCharacterizationForModelVariable(m)$
        else
            $V \leftarrow T'$
        end
        $e \leftarrow$ new value map entry $e$ with
              $key(e) \leftarrow m$ and $value(e) \leftarrow res$
        $parent(e) \leftarrow V$
        $valueMap(f) \leftarrow valueMap(f) \cup \{e\}$
    end
end
```

Algorithm 4: Processing a ModelVariable with Relationships

whose parent attribute is set accordingly, i.e., set to an empty set if m has been characterized without influencing parameters, or set to the value map entries T' if the value for m was successfully obtained using method *getCharacterizationForParameterDependency*.

After method *ProcessModelVariableWithRelationships* has been executed, the following postcondition holds: For the given model variable m there is now one value map entry e in the given stack frame. The parent property of e is initialized, but may refer to an empty set. Note that Algorithm 4 does not consider characterization type IDENTITY for relationships. This characterization type is a simple special case of characterization type EXPLICIT and therefore omitted.

In the following, we describe method *ResolveNullValuesTransitively* shown in Algorithm 5. As mentioned above, for value map entries with a null value, this method traces back existing dependency propagation relationships in order to search for independent parameters where non-null values are available.

```
1  ResolveNullValuesTransitively($C$ : Set(ValueMapEntry))
2  begin
3      $T \leftarrow C$
4      while true do
5          if $\neg\exists t \in T : (value(t) = NULL)$ then
6              return $T$
7          else
8              $D \leftarrow \emptyset$
9              $A \leftarrow \emptyset$
10             foreach $t \in T$ with $value(t) = NULL$ do
11                 $D \leftarrow D \cup \{t\}$
12                 /* ValueMapEntries with value = NULL stem only from
                      DependencyPropagationRelationships with EMPIRICAL
                      characterization.  Jump to their origin and check if
                      the origin has a non-null value.                      */
13                 $A \leftarrow A \cup parent(t)$
14             end
15             $T \leftarrow (T \setminus D) \cup A$
16             if $A = \emptyset$ then
17                 return $T$
18             end
19         end
20     end
21 end
```

Algorithm 5: Traversal of Relationships with EMPIRICAL Characterization

The method starts with an initialization of helper set T with method input parameter V which is a set of value map entries. A loop follows. In case the helper set contains only non-null value map entries, the loop terminates and the helper set is returned as result. Otherwise, each value map entry $t \in T$ with value $NULL$ is removed from T and each value map entry in *parent(t)* is added to T. If no value map entry is added to T, the loop terminates and T is returned as result.

The loop always terminates because cycles of relationships are not allowed (see Section 4.1.5.3). The loop terminates at the latest if T contains value map entries of all InfluencingParameters whose number is finite. Then, a loop iteration cannot add more value map entries to T and the loop termination condition in line 16 of Algorithm 5 holds.

5.2.2.3 Example

The following example illustrates how relationships are resolved and how model variable characterizations are obtained. Figure 5.5 shows three components with influencing parameters and several relationships. Component instance *A* provides a service *a* with influencing parameters *a1* and *a2*. It requires two component external services namely service *b* with influencing parameter *b1* and service *c* with influencing parameter *c1*, provided by component instances *B* and *C*, respectively. *B* again requires service *c* with parameter *c1*. The service behavior model of service *c* contains an influenced variable referring to a resource demand *rd*. Note that the service behavior models for the provided services are omitted in the diagram. We denote the service behaviors in the following as *sba_a*,*sba_b* and *sba_c* for service *a*, *b* and *c*.

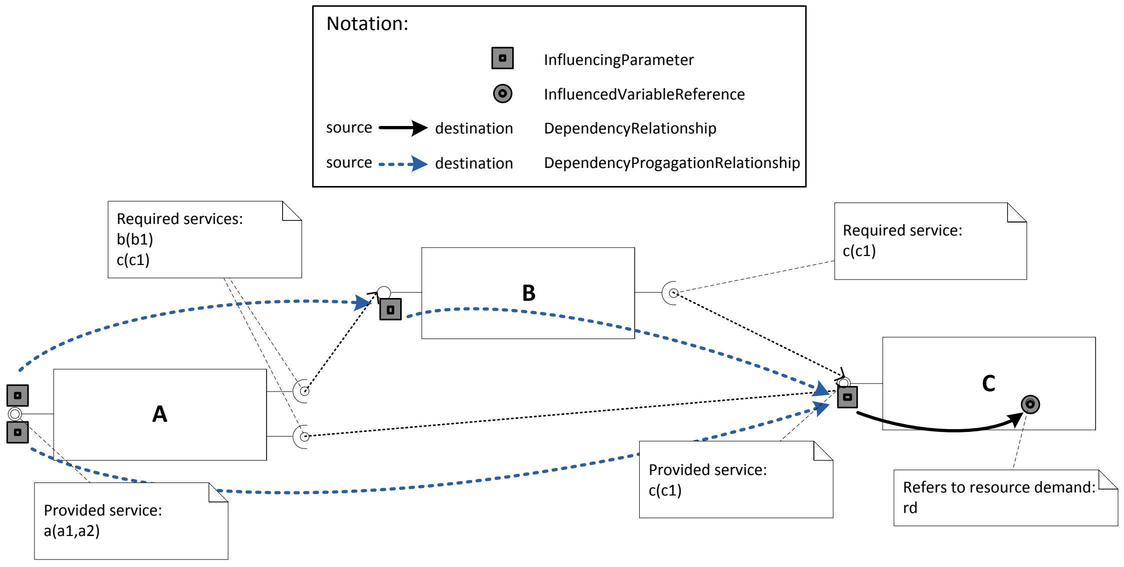

Notation:
InfluencingParameter
InfluencedVariableReference
source destination DependencyRelationship
source destination DependencyProgagationRelationship
Required services: b(b1) c(c1)
Required service: c(c1)
B
A
C
Provided service: a(a1,a2)
Provided service: c(c1)
Refers to resource demand: rd

Figure 5.5: Example: Dependency Relationship Solving

The figure furthermore shows how the influencing parameters are connected with each other. There are dependency propagation relationships between *a1* and *b1*, *b1* and *c1* as well as between *a2* and *c1*. There is a dependency relationship between *c1* and the influenced variable reference that refers to *rd*. We assume that all mentioned relationships are characterized as EMPIRICAL. For the meta-model of relationships between model variables and their characterization, see Section 4.1.5.3.

For the pre-processing, we assume there is a call to service *a* that injects a value x for influencing parameter *a1*. Furthermore, we assume that monitoring method *getCharacterizationForModelVariable* provides characterizations for resource demands, response times and control flow variables. Otherwise, a performance prediction would not be possible. For influencing parameters it returns *NULL*.

In the following, it is described how value x of parameter *a1* is propagated through the components in order to obtain a value for resource demand *rd* depending on value x. The process is described by showing the output model step-by-step. The output model is a callstack, an instance of the meta-model shown in Figure 5.4. We illustrate stack frame instances using the notation depicted in Figure 5.6. A StackFrame is shown as a box consisting of three compartments: (i) The lower compartment shows the value of property serviceBehavior of a modelStackFrame, (ii) the compartment in the middle shows the ValueMapEntries belonging to a StackFrame, and (iii) the upper compartment shows the Successors of a StackFrame. Associations from a Successor to another StackFrame as

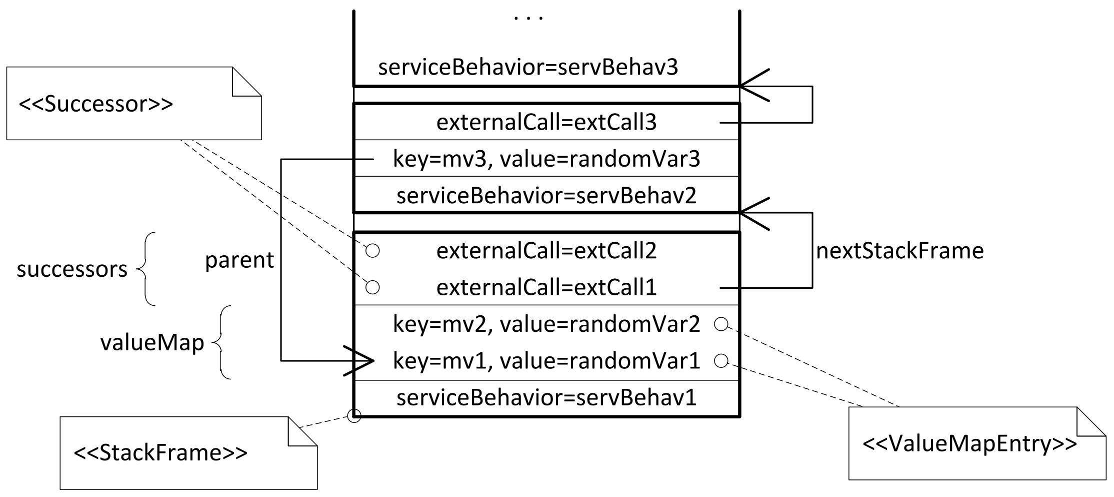

...
serviceBehavior=servBehav3
<<Successor>>
externalCall=extCall3
key=mv3, value=randomVar3
serviceBehavior=servBehav2
successors
parent
externalCall=extCall2
externalCall=extCall1
nextStackFrame
valueMap
key=mv2, value=randomVar2
key=mv1, value=randomVar1
serviceBehavior=servBehav1
<<StackFrame>>
<<ValueMapEntry>>

Figure 5.6: Notation for Stack Frame Instances

well as associations from a ValueMapEntry to another ValueMapEntry are shown as arrows referring to the corresponding StackFrame respectively ValueMapEntry.

Figure 5.7 shows the process of parameter dependency resolution and model parameterization in six steps. The process is invoked by calling method *Solve* of Algorithm 2 for *sba_a* with a stack frame as parameter that contains an injected value map entry for *a1*.

- Figure 5.7a shows the result when all model variables of *sba_a* have been processed. The value of influencing parameter *a1* is assumed to be x. The value of influencing parameter *a2* is obtained using method call *getCharacterizationForModelVariable*$(a2)$ which returns *NULL*. Then, the first external call in *sba_a* leads to an invocation of method *Solve* for *sba_b*.
- Figure 5.7b shows the result when all model variables of *sba_b* have been processed. The value of influencing parameter *b1* is tried to be resolved using the relationship between *a1* and *b1*, but method call *getCharacterizationForParameterDependency*$(\{a1 = x\}, b1)$, where $\{a1 = x\}$ represents the value map entry for *a1* in the first stack frame, returns *NULL*. Then, the external call in *sba_b* leads to an invocation of method *Solve* for *sba_c*.
- Figure 5.7c shows the stack frames when influencing parameter *c1* has been processed. Note that there are two relationships pointing to *c1*. The independent parameters of the two relationships between *a2* and *c1* as well as *b1* and *c1* both have the value *NULL*, however, method *ResolveNullValuesTransitively* finds out that the value map entry for *b1* refers to a parent value map entry with a non-null value. Given that the involved relationships are marked as empirically characterized, *c1* and *a1* are considered to be related. The value for *c1* is tried to be obtained by calling *getCharacterizationForParameterDependency*$(\{a1 = x\}, c1)$ which returns *NULL*.
- Figure 5.7d shows the stack frame state when model variable *rd* has been processed. Following the relation between *rd* and *c1*, and between the value map entries of *c1* and *a1*, the value for *rd* is tried to be obtained with *getCharacterizationForParameterDependency*$(\{a1 = x\}, rd)$. This time, the function evaluates to y which is stored as value in the third stack frame.

Having all model variables in *sba_c* processed, the methods *Solve* for *sba_c* and *Solve* for *sba_b* end. Thus, the external call in service behavior *sba_a* to service *b* is processed, too. However, method *Solve* for *sba_a* has another external call to service *c* which is processed

Step	Calls to Interface *IApplicationLevelMonitor*	Result
1	$getCharacterizationForModelVariable(a2)$	$NULL$
2	$getCharacterizationForParameterDependency(\{a1 = x\}, b1)$	$NULL$
3	$getCharacterizationForParameterDependency(\{a1 = x\}, c1)$	$NULL$
4	$getCharacterizationForParameterDependency(\{a1 = x\}, rd)$	y
5	$getCharacterizationForModelVariable(c1)$	$NULL$
6	$getCharacterizationForModelVariable(rd)$	y'

Table 5.1: Example: Calls to Monitoring Infrastructure

afterwards. Therefore, a new stack frame is created and method *Solve* is called for service behavior *sba_c*. Note that there are now two stack frames for *sba_c*, one stack frame for each call path to service *c*.

- Figure 5.7e shows the stack frames when influencing parameter *c1* has been processed in the second stack frame of service behavior *sba_c*. Since the value map entry of *a2* and all of its parents have a *NULL* value, the value for *c1* is obtained using $getCharacterizationForParameterDependency(\{\}, c1)$ which is equivalent to $getCharacterizationForModelVariable(c1)$ and evaluates to *NULL*.
- Figure 5.7f shows the state when model variable *rd* has been processed. This time, the value for *rd* is obtained using $getCharacterizationForParameterDependency(\{\}, rd) = getCharacterizationForModelVariable(rd)$ which evaluates to y'.

Note that model variable *rd* has two values, depending on the call path. For the call path from *sba_a* to *sba_b* to *sba_c*, the relationships with empirical characterization are transitively resolved which results in the value $y = getCharacterizationForParameterDependency(rd, \{a1 = x\})$, where y is the conditional probability distribution for rd depending on value x. For the call path from *sba_a* to *sba_c*, no relationships can be used for a characterization, and thus the fallback value $y' = getCharacterizationForModelVariable(rd)$ is used.

Overall, the algorithm issues a list of calls to the monitoring infrastructure via interface *IApplicationLevelMonitor* as shown in Table 5.1. The monitoring infrastructure thus returns only values for requests where resource demand *rd* is involved. The other way round, if a value for *a1* is injected and the solving algorithm is triggered, the monitoring infrastructure can notice that the only parameter dependencies that need to be monitored are the parameter dependencies where the dependent variable is either a resource demand, a response time, or a control flow variable. In the example, there is such a parameter dependency between *rd* and *a1*. Thus, to determine the relevant parameter dependencies, the monitoring infrastructure does not need to search the model instance for relationships itself, it is the solving algorithm that can be used to obtain the relevant parameter dependencies that need to be observed. This approach is used later in Chapter 6, where it is described how model parameters as well as parameter dependencies are monitored.

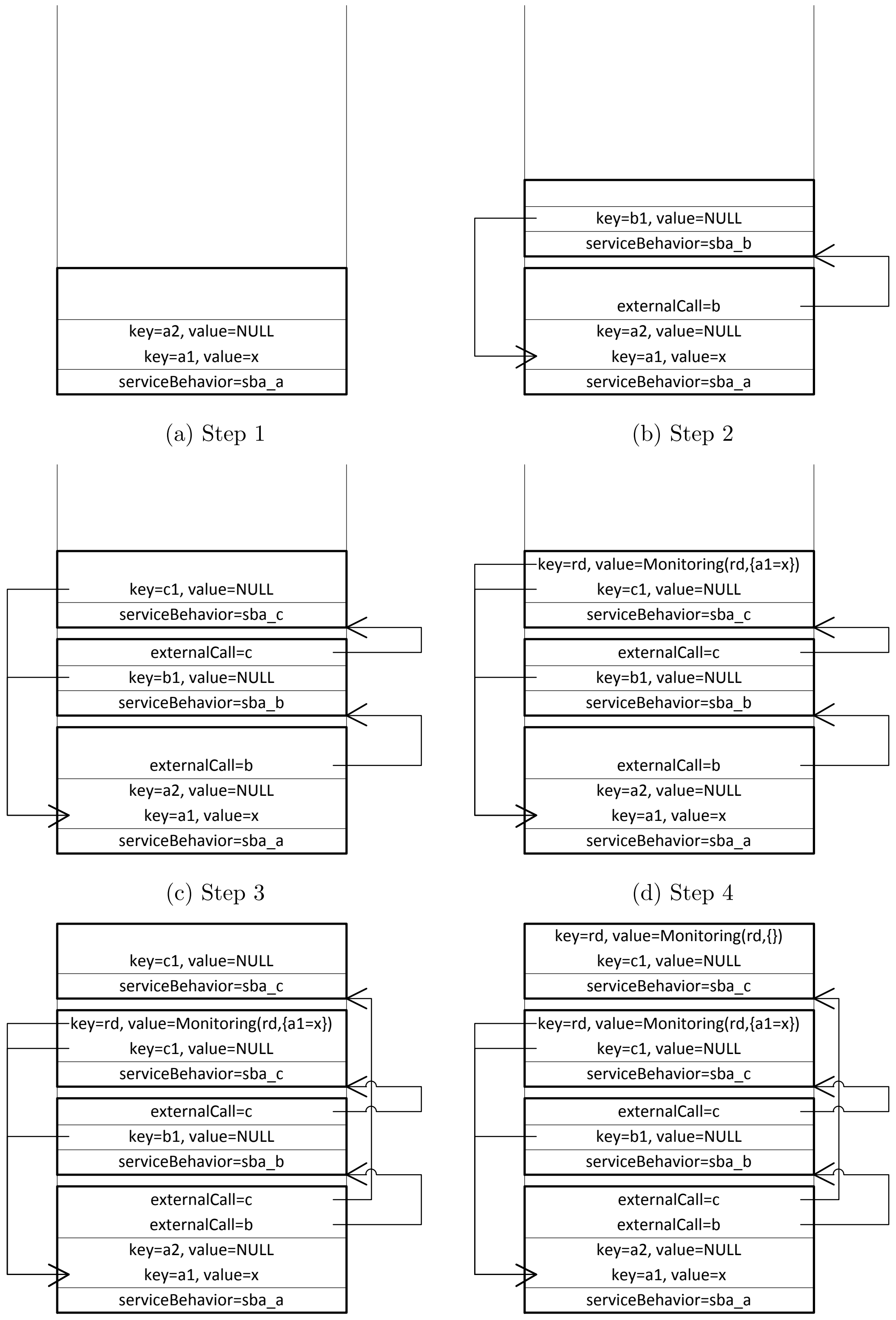
5.2.3 Complexity

In the following, we estimate the complexity of the described algorithms in Big-O-Notation.

Method *ResolveNullValuesTransitively* in Algorithm 5 has a complexity of

$$O(ResolveNullValuesTransitively) = totalN_{inflP},$$

since in the worst case the loop may iterate until the result set equals to the total number of influencing parameters $totalN_{inflP}$ in the model instance.



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

(e) Step 5 (f) Step 6

Figure 5.7: Example: Resolution of Relationships Step-By-Step

Method *ProcessModelVariableWithRelationships* in Algorithm 4 consists of two parts. The first part deals with relationships with characterization EXPLICIT, the second part deals with relationships with characterization EMPIRICAL. In both parts, the number of influencing parameters that are independent parameters of the processed relationships have an influence on the part's complexity. Obviously, this number is bounded by the total number of influencing parameters $TotalN_{inflP}$. For each influencing parameter, the corresponding value map entry has to be found in the path of stack frames whose maximum length is denoted as $maxlength_{callpath}$. For the first part, recall that at most one explicitly characterized relationship may point to a designated model variable, the worst-case complexity is then given by

$$totalN_{inflP} * maxlength_{callpath} + O(Calculate(totalN_{inflP})),$$

where $O(Calculate(totalN_{inflP})) = totalN_{inflP} * O(klog(k))$ and $O(klog(k))$ is the complexity for a discrete convolution using Fast Fourier transformation, where k is "the number of sampling points in the involved distribution function" (Koziolek, 2008, p.141). For the second part, the worst-case complexity is given by

$$\begin{gathered} totalN_{inflP} * maxlength_{callpath} + O(ResolveNullValuesTransitively) \\ +O(getCharacterizationForParameterDependency(totalN_{inflP})). \end{gathered}$$

Thus, for one method call *ProcessModelVariableWithRelationships*, the worst-case complexity is

$$\begin{gathered} totalN_{inflP} * (2 * maxlength_{callpath} + O(klog(k)) + 1) \\ +O(getCharacterizationForParameterDependency(totalN_{inflP})). \end{gathered}$$

Assuming that $k \geq maxlength_{callpath}$, we can simplify the expression in Big-O-Notation to

$$totalN_{inflP} * O(klog(k)) + O(getCharacterizationForParameterDependency(totalN_{inflP})),$$

in other words, the complexity is dominated by the calculation and the monitoring data access to obtain the resulting value dependent on multiple influencing parameters.

For one method call *ProcessModelVariables* (see Algorithm 3), an upper bound for the complexity is

$$totalN_{inflP} * (totalN_{rel} + O(ProcessModelVariableWithRelationships)) + totalN_{mv}$$

where $totalN_{mv}$ is the total number of model variables and $totalN_{rel}$ the total number of relationships.

Method *ProcessModelVariables* is called whenever method *Solve* is invoked on a service behavior abstraction. A minimal bound for the number of *Solve* invocations is $totalN_{extCall}$, the total number of external calls. The upper bound is theoretically of exponential complexity, because the number of call paths can be of exponential complexity. Assume an artificial example that consists of concatenations of the component compositions depicted in Figure 5.5, i.e., there are several component triples A_i, B_i, C_i connected like the components A, B, C; and C_i is connected to A_{i+1} via an external call. Then the number of call paths is in $O(2^{totalN_{extCall}})$ because each component triple has two different paths from A_i to C_i, namely A_i to B_i to C_i as well as A_i directly to C_i. However, note that this is an artificial example. In our case studies in Chapter 7, paths in component compositions were typically unique, leading to a linear dependency between the number of external calls and calls to method *Solve*.

5.3 Model Solving

The *tailored* model solving step shown in Figure 5.1 uses existing model solving techniques based on established modeling formalisms. In this section, the focus is on the model solving techniques. The *tailoring mechanisms* are described in Section 5.4.

There are basically two types of model solving techniques, namely simulative and analytical solving. A simulation can provide the best prediction accuracy but has the lowest prediction speed compared to other solving techniques. Complex performance metrics such as response time distributions can be provided. Analytical model solvers typically have lower prediction overhead compared to simulation, but they are often restricted in terms of predictable performance metrics and also model input parameters. Analytical solvers such as LQNS (Franks et al., 1996) are often restricted to exponentially distributed resource demands, delays and inter-arrival times (Balsamo et al., 2004), or have limited capabilities to analyze blocking behavior or simultaneous resource possession (Menasce and Virgilio, 2000; Woodside et al., 2006; Gilmore et al., 2005). Since in our context, model variables are typically characterized with empirical distributions obtained from monitoring data, the assumptions of exponentially distributed service times and inter-arrival times often do not hold. However, analytical solving techniques using exponential distributions instead of empirical distributions can still provide approximate results.

As simulation technique, we transform a Descartes Modeling Language (DML) instance to a QPN (Bause, 1993) and simulate it using the simulation engine SimQPN (Kounev and Buchmann, 2006) (see Section 5.3.3). QPNs have been used successfully to model several different classes of distributed systems (Kounev, 2006; Kounev et al., 2011, 2008; Nou et al., 2009). SimQPN is an established simulator for QPNs (Kounev and Buchmann, 2006; Spinner et al., 2012) that provides fine-grained options to control what type and amount of data is logged during the simulation run. The more data is logged, the longer the simulation run takes. Hence, the logging configuration can be used to tailor the simulation to the given performance query.

As analytical solving technique, we make use of the established analytical solver tool LQNS (Franks et al., 2011, 1996, 2009). The LQNS solver implements several analytical solving techniques such as MVA (Bolch et al., 1998) and combines the advantages of other existing solvers, namely SRVN (Woodside et al., 1995) and MOL (Rolia and Sevcik, 1995). Given that LQNS is a solver for LQNs, transformations from DML instances to LQNs have to be provided.

Furthermore, we apply an analytical solving technique called asymptotic bounds analysis (Bolch et al., 1998) (see Section 5.3.1). Bounds analysis is fast because it works with several approximations that cause inaccuracies, however, it can be good enough to make quick decisions, when approximative performance results are sufficient (Menascé et al., 1994; Bolch et al., 1998; Menascé et al., 2004b).

5.3.1 Transformation to Queueing Petri Nets

5.3.1.1 Queueing Petri Nets

Queueing Petri Nets (QPNs) can be seen as a combination of a number of different extensions to conventional Petri Nets (PNs) along several different dimensions. In this section, we briefly discuss how QPNs have evolved from PNs. A more detailed discussion including formal definitions can be found in Bause (1993).

An ordinary PN is a bipartite directed graph composed of places, drawn as circles, and transitions, drawn as bars (Bause and Kritzinger, 2002). Different extensions to ordinary

PNs have been developed in order to increase the modeling convenience and/or the modeling power. Colored PNs (CPNs) allow a type (color) to be attached to a token. A color function assigns a set of colors to each place, specifying the types of tokens that can reside in the place. In addition to introducing token colors, CPNs also allow transitions to fire in different *modes* (transition colors).

Other extensions to ordinary PNs allow temporal (timing) aspects to be integrated into the net description. In particular, Stochastic Petri Nets (SPNs) attach an exponentially distributed *firing delay* to each transition, which specifies the time the transition waits after being enabled before it fires. Generalized Stochastic PNs (GSPNs) allow two types of transitions to be used: immediate and timed. Once enabled, immediate transitions fire in zero time. If several immediate transitions are enabled at the same time, the next transition to fire is chosen based on *firing weights* (probabilities) assigned to the transitions. Timed transitions fire after a random exponentially distributed firing delay as in the case of SPNs. The firing of immediate transitions always has priority over that of timed transitions. Combining CPNs and GSPNs leads to Colored GSPNs (CGSPNs).

While CGSPNs have proven to be a very powerful modeling formalism, they do not provide any means for direct representation of queueing disciplines (scheduling strategies). The attempts to eliminate this disadvantage have led to the emergence of QPNs which add queueing and timing aspects to the places of CGSPNs. This is done by allowing queues (service stations) to be integrated into places of CGSPNs. A place of a CGSPN that has an integrated queue is called a *queueing place* and consists of two components, the *queue* and a *depository* for tokens which have completed their service at the queue. The queue is defined by its scheduling strategy, number of servers and service time distribution where the latter can be specified on a per token color basis.

The behavior of the net is as follows: tokens, when fired into a queueing place by any of its input transitions, are inserted into the queue according to the queue's scheduling strategy. The time tokens spend in the queue includes the time spent waiting for service and the time receiving service which depends on the queue's service time distribution. While residing in the queue, tokens are not available for output transitions of the queueing place. After completion of its service at the queue, a token is immediately moved to the depository, where it becomes available for output transitions of the place. This type of queueing place is called *timed* queueing place.

In addition to timed queueing places, QPNs also introduce *immediate* queueing places, which allow pure scheduling aspects to be described. Tokens in immediate queueing places can be viewed as being served immediately. The rest of the net behaves like a normal CGSPN. QPNs also support so-called *subnet places* that contain nested QPNs. Figure 5.8 shows the notation used for ordinary places, queueing places, and subnet places.

We use the following formal definition of QPNs which is based on Bause (1993); Kounev and Buchmann (2006). For this definition of QPNs, with the Queueing Petri Net Modeling Environment (QPME) tool chain (Spinner et al., 2012) there is mature tool support available. Note that we omit the concept of timed transitions, because timed transitions can be replaced with immediate transitions and immediate queueing places.

Definition. *A QPN is an eight-tuple* (P, T, C, Q, F, I, D, W) *where:*

1. $P = \{p_1, \ldots, p_k\}$ *is a finite set of places.*
2. $T = \{t_1, \ldots, t_l\}$ *is a finite set of transitions,* $P \cap T = \emptyset$*.*
3. $C = \{c_1, \ldots, c_m\}$ *is a finite set of colors.*
4. $Q = \{q_1, \ldots, q_n\}$ *is a finite set of queues.*

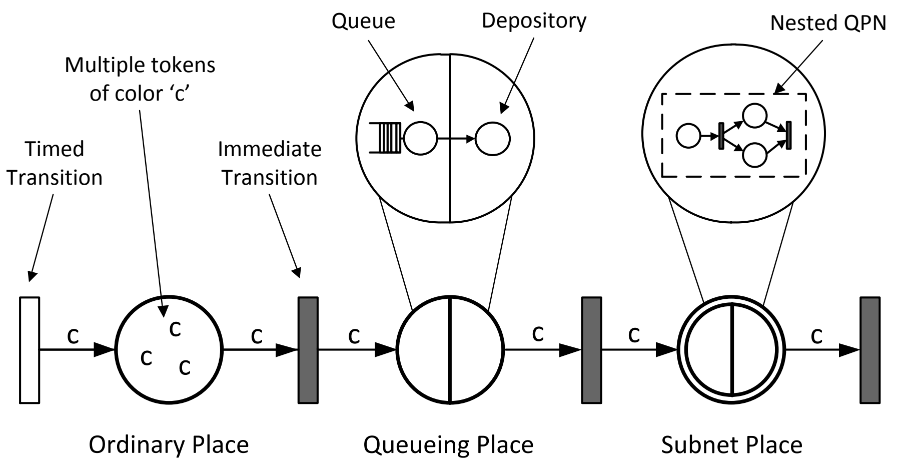


Figure 5.8: QPN Notation (Meier et al., 2011)

5. *$F = (F_M, F_C, F_C^0)$ where*
   - *$F_C \in [P \to \mathcal{P}(C)]$[1] is a function that assigns a set of colors to each place,*
   - *F_M is a function that assigns a finite set of modes M_t to each transition $t \in T$, i.e., $\forall t \in T : F_M(t) = M_t = \{m_{t,1}, \dots, m_{t,t_n}\}$. Each mode is assigned to a unique transition, i.e., $\forall t_i, t_j : t_i \neq t_j \implies F_M(t_i) \cap F_M(t_j) = \emptyset$,*
   - *F_C^0 is a function defined on P describing the initial marking, such that $F_C^0(p) \in F_C(p)_{MS}$[2].*
6. *$I = (I^-, I^+)$ where I^- and I^+ are the backward and forward incidence functions defined on $P \times T$, such that $I^-(p,t), I^+(p,t) \in [F_M(t) \to F_C(p)_{MS}]$, $\forall (p,t) \in P \times T$.*
7. *$D = (\tilde{Q}_1, \tilde{Q}_2, \tilde{Q}_P, D_Q)$ where*
   - *$\tilde{Q}_1 \subseteq P$ is the set of timed queueing places,*
   - *$\tilde{Q}_2 \subseteq P$ is the set of immediate queueing places, $\tilde{Q}_1 \cap \tilde{Q}_2 = \emptyset$,*
   - *$\tilde{Q}_P \in [\tilde{Q}_1 \cup \tilde{Q}_2 \to Q]$ is a function that assigns a queue to a queueing place p,*
   - *D_Q is a function that assigns a queue description[3] to a queue $q \in Q$. The queue description $D_Q(q)$ has to take all colors $F_C(p)$ into consideration where $\tilde{Q}_P(p) = q$. In the following, we use the expression $(D_Q(q))(demand(c))$ to denote the service demand for tokens of color c for queue $q \in Q$.*
8. *W is a function on T that assigns each transition $tinT$ a function $W(t) \in [F_M(t) \to \mathbb{R}^+]$ such that $\forall m \in F_M(t) : (W(t))(m) \in \mathbb{R}^+$ is interpreted as a firing weight specifying the relative firing frequency of mode m in transition t.*

Queueing places are normally used to model system resources, e.g., CPUs, disk drives, or network links. Tokens in the QPN are used to model requests or transactions processed by the system. Arriving tokens at a queueing place are first served at the queue and then

[1]$\mathcal{P}$ denotes power sets. $\mathcal{P}(C)$ denotes the set of all subsets of C.

[2]The subscript MS denotes multisets. $F_C(p)_{MS}$ denotes the set of all finite multisets of $F_C(p)$.

[3]In the most general definition of QPNs, queues are defined in a very generic way allowing the specification of arbitrarily complex scheduling strategies taking into account the state of both the queue and the depository of the queueing place (Bause, 1993). We use conventional queues as defined in queueing network theory (Bolch et al., 1998).

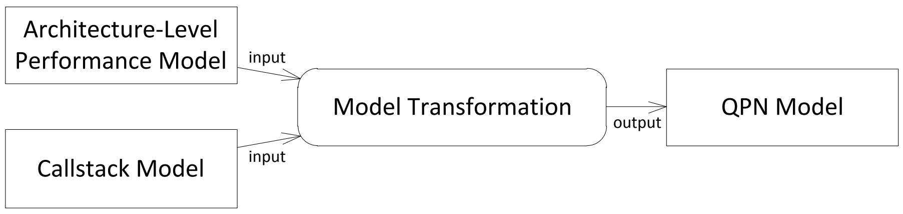


Figure 5.9: Input and Output of Transformation to QPNs

they become available for firing of output transitions. When a transition fires it removes tokens from some places and creates tokens at others. Usually, tokens are moved between places representing the flow-of-control during transaction processing. QPNs have been used successfully to model several different classes of distributed systems (Kounev, 2006; Kounev et al., 2011, 2008; Nou et al., 2009).

5.3.1.2 Transformation

In this section, we describe how we transform the service behavior models presented in Chapter 4 to QPNs. As shown in Figure 5.9, the input of the transformation is the architecture-level performance model itself and the callstack model, which is the output of the pre-processing step described in Section 5.2. The callstack model indicates the sequence of service behaviors together with values of contained model variables. Here, we first describe the transformation. The tailoring strategy is described later in Section 5.4.

The transformation starts with the usage profile model (see Section 4.3) and then traverses the application architecture guided by the callstack model.

- For each UsageScenario the transformation traverses the ScenarioBehavior.
- Traversing the ScenarioBehavior means transforming each user action to a QPN representation. Whenever a SystemCallUserAction is reached, the callstack of stack frames is used to navigate to the corresponding ServiceBehaviorAbstraction.
- The traversal of a ServiceBehaviorAbstraction is similar. Each behavior abstraction is transformed to a QPN representation. Whenever an ExternalCall is reached, the stack frames are used to navigate to the next ServiceBehaviorAbstraction. Whenever a ModelVariable is accessed, its value can be read from the current stack frame.

The traversal involves the transformation of the following modeling constructs:

- Open workload (model entity OpenWorkloadType)
- Closed workload (model entity ClosedWorkloadType)
- Calls (model entities SystemCallUserAction, ExternalCall)
- Branches (model entities BranchUserAction, BranchAction)
- Loops (model entities LoopUserAction, LoopAction, ExternalCallFrequency)
- Forks with and without synchronization barrier (model entity ForkAction)
- Acquire and release passive resource (model entities AcquireAction and ReleaseAction)
- Response Times (model entity ResponseTime and DelayUserAction)
- Resource Demands (model entity ResourceDemand)

Here we do not distinguish between FineGrainedBehavior, CoarseGrainedBehavior and BlackBoxBehavior but consider individual modeling constructs. For instance, model entity ResourceDemand can be found in both the FineGrainedBehavior and the CoarseGrainedBehavior abstractions. For each of the listed modeling constructs, we now describe its transformation to QPNs. The transformation is initialized with an empty net $output = (P, T, C, Q, F, I, D, W)$ with $P := \emptyset, T := \emptyset, C := \emptyset, Q := \emptyset$. All queues $q \in Q$ shall accept general distributions as resource demands. Given that in our context, we obtain resource demands and response times as empirical distributions from monitoring data, assumptions about their distributions would not hold.

The transformation is based on the informal description presented in Meier et al. (2011); Meier (2010) and the more in-depth analysis in Brosig et al. (2014). We provide a formalization of the transformation using the definition of QPNs given above. In Koziolek (2008), there is a transformation from PCM Resource Demanding Service Effect Specifications (RDSEFFs) to "an extended form of QPNs which is not supported by available tools. It uses tokens that carry arbitrary properties instead of just a color value." (Meier et al., 2011, p.2). Thus, tokens of the same color code can be distinguished which is not possible in standard QPNs as defined here and supported by available modeling tools.

Open Workload

An open workload represents incoming requests that arrive with a specified inter-arrival time. See Figure 5.10 for an illustration of how an open workload is modeled in as QPN.

A request is translated to a token of color c that is unique for this request type, i.e., unique for a ScenarioBehavior. There is an immediate queueing place *Client-Place* that serves as token generator for tokens of color c. The initial number of tokens in the *Client-Place* is set to one, the service demand of c at the *Client-Place* is set to the request inter-arrival time specified in the usage profile model. Transition *Client-Entry* has a mode that fires a token from the *Client-Place* to the *Behavior-Begin-Place*, and furthermore adds a new token of color c to the *Client-Place*. Transition *Client-Entry* consumes a token of color c from the *Behavior-End-Place* but does not propagate it, i.e., the transition *destroys* a token of color c. The two places *Behavior-Begin-Place* and *Behavior-End-Place* denote the begin and end of client behavior descriptions.

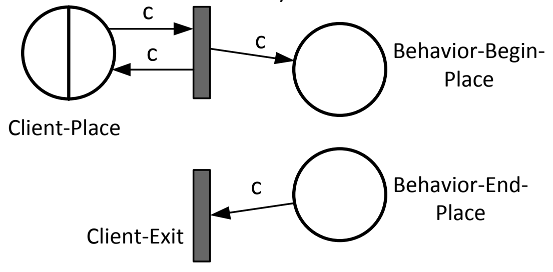


Figure 5.10: QPN Representation of Open Workload, cf. Meier (2010)

Formally, an open workload is transformed to a QPN with:

$$
\begin{aligned}
&P = P \cup \{p_{\textit{Client-Place}}, p_{\textit{Behavior-Begin-Place}}, p_{\textit{Behavior-End-Place}}\}, \tilde{Q}_2 = \tilde{Q}_2 \cup \{p_{\textit{Client-Place}}\}\\
&T = T \cup \{t_{\textit{Client-Entry}}, t_{\textit{Client-Exit}}\}\\
&C = C \cup \{c\}\\
&Q = Q \cup \{q\}\\
&F_C(p_{\textit{Client-Place}}) = \{c\}, F_C(p_{\textit{Behavior-Begin-Place}}) = \{c\}, F_C(p_{\textit{Behavior-End-Place}}) = \{c\}\\
&F_M(t_{\textit{Client-Entry}}) = \{m_{t_{\textit{Client-Entry}},1}\}\\
&F_M(t_{\textit{Client-Exit}}) = \{m_{t_{\textit{Client-Exit}},1}\}\\
&F_C^0(p_{\textit{Client-Place}}) = \{c\}_{MS}\\
&(I^-(p_{\textit{Client-Place}}, t_{\textit{Client-Entry}}))(m_{t_{\textit{Client-Entry}},1}) = \{c\}_{MS}\\
&(I^+(p_{\textit{Client-Place}}, t_{\textit{Client-Entry}}))(m_{t_{\textit{Client-Entry}},1}) = \{c\}_{MS}\\
&(I^+(p_{\textit{Behavior-Begin-Place}}, t_{\textit{Client-Entry}}))(m_{t_{\textit{Client-Entry}},1}) = \{c\}_{MS}\\
&(I^-(p_{\textit{Behavior-End-Place}}, t_{\textit{Client-Exit}}))(m_{t_{\textit{Client-Exit}},1}) = \{c\}_{MS}\\
&\tilde{Q}_P(p_{\textit{Client-Place}}) = \{q\}\\
&(D_Q(q))(demand(c)) = \textit{request\_inter-arrival\_time}
\end{aligned}
$$

Closed Workload

A closed workload represents a fixed number of clients that repeatedly issue system requests with a certain think time between two requests. An illustration of how a closed workload is modeled in a QPN is shown in Figure 5.11.

A request is translated to a token of color c that is unique for the type of the request, i.e., unique for a ScenarioBehavior. There is an immediate queueing place *Client-Place* that serves as repository for tokens of color c. The initial number of tokens in the *Client-Place* is set to the specified number of clients, the service demand of c at the *Client-Place* is set to the specified think time. Transition *Client-Entry* has a mode that propagates a token from the *Client-Place* to the *Behavior-Begin-Place*. Transition *Client-Exit* has a mode that propagates a token from the *Behavior-End-Place* back to the *Client-Place*.

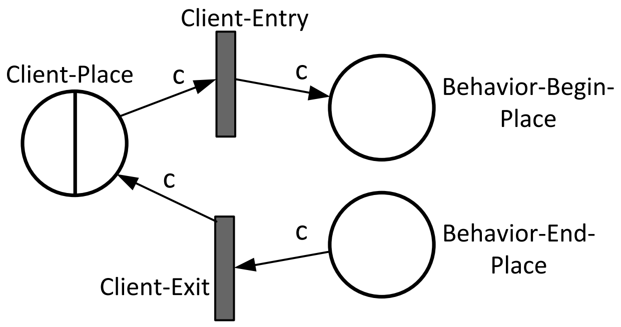


Figure 5.11: QPN Representation of Closed Workload, cf. Meier (2010)

Formally, a closed workload is transformed to a QPN with:

$P = P \cup \{p_{\textit{Client-Place}}, p_{\textit{Behavior-Begin-Place}}, p_{\textit{Behavior-End-Place}}\}, \tilde{Q}_2 = \tilde{Q}_2 \cup \{p_{\textit{Client-Place}}\}$
$T = T \cup \{t_{\textit{Client-Entry}}, t_{\textit{Client-Exit}}\}$
$C = C \cup \{c\}$
$Q = Q \cup \{q\}$
$F_C(p_{\textit{Client-Place}}) = \{c\}, F_C(p_{\textit{Behavior-Begin-Place}}) = \{c\}, F_C(p_{\textit{Behavior-End-Place}}) = \{c\}$
$F_M(t_{\textit{Client-Entry}}) = \{m_{t_{\textit{Client-Entry}},1}\}$
$F_M(t_{\textit{Client-Exit}}) = \{m_{t_{\textit{Client-Exit}},1}\}$
$F_C^0(p_{\textit{Client-Place}}) = \{ \underbrace{c, \ldots, c}_{\text{number of clients}} \}_{MS}$
$(I^-(p_{\textit{Client-Place}}, t_{\textit{Client-Entry}}))(m_{t_{\textit{Client-Entry}},1}) = \{c\}_{MS}$
$(I^+(p_{\textit{Behavior-Begin-Place}}, t_{\textit{Client-Entry}}))(m_{t_{\textit{Client-Entry}},1}) = \{c\}_{MS}$
$(I^-(p_{\textit{Behavior-End-Place}}, t_{\textit{Client-Exit}}))(m_{t_{\textit{Client-Exit}},1}) = \{c\}_{MS}$
$(I^+(p_{\textit{Client-Place}}, t_{\textit{Client-Exit}}))(m_{t_{\textit{Client-Exit}},1}) = \{c\}_{MS}$
$\tilde{Q}_P(p_{\textit{Client-Place}}) = \{q\}$
$(D_Q(q))(\textit{demand}(c)) = \textit{think\_time}$

Calls

A call, e.g., a SystemCallUserAction or an ExternalCall can be represented in a QPN as depicted in Figure 5.12. The source of the call is represented by two places *Call-Begin-Place* and *Call-Exit-Place*. The target of the call is a ServiceBehaviorAbstraction whose start and end is represented by the places *Behavior-Begin-Place* and *Behavior-End-Place*, respectively. Transition *Call-Entry* propagates a token from the *Call-Begin-Place* to the *Behavior-Begin-Place*. Transition *Call-Exit* then propagates a token from the *Behavior-End-Place* to the *Call-End-Place*.

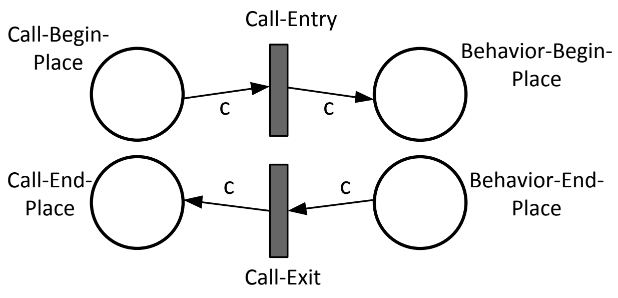


Figure 5.12: QPN Representation of Calls, cf. Meier (2010)

Formally, a call in the source model is transformed to a QPN with:

$P = P \cup \{p_{\textit{Call-Begin-Place}}, p_{\textit{Call-End-Place}}, p_{\textit{Behavior-Begin-Place}}, p_{\textit{Behavior-End-Place}}\}$
$T = T \cup \{t_{\textit{Call-Entry}}, t_{\textit{Call-Exit}}\}$
$F_C(p_{\textit{Call-Begin-Place}}) = \{c\}, F_C(p_{\textit{Call-End-Place}}) = \{c\}$
$F_C(p_{\textit{Behavior-Begin-Place}}) = \{c\}, F_C(p_{\textit{Behavior-End-Place}}) = \{c\}$
$F_M(t_{\textit{Call-Entry}}) = \{m_{t_{\textit{Call-Entry}},1}\}$
$F_M(t_{\textit{Call-Exit}}) = \{m_{t_{\textit{Call-Exit}},1}\}$
$(I^-(p_{\textit{Call-Begin-Place}}, t_{\textit{Call-Entry}}))(m_{t_{\textit{Call-Entry}},1}) = \{c\}_{MS}$
$(I^+(p_{\textit{Behavior-Begin-Place}}, t_{\textit{Call-Entry}}))(m_{t_{\textit{Call-Entry}},1}) = \{c\}_{MS}$
$(I^-(p_{\textit{Behavior-End-Place}}, t_{\textit{Call-Exit}}))(m_{t_{\textit{Call-Exit}},1}) = \{c\}_{MS}$
$(I^+(p_{\textit{Call-End-Place}}, t_{\textit{Call-Exit}}))(m_{t_{\textit{Call-Exit}},1}) = \{c\}_{MS}$

Branches

Figure 5.13 shows a branch modeled in a QPN. The begin of the branch is represented by the place *Branch-Begin-Place*, the end of the branch is represented by the place *Branch-End-*

Place. For each of the N branch transitions (not to be confused with a QPN transition) there is a place for the begin and end of the transition's behavior. Place *Behaviori-Begin-Place* denotes a transition's begin while *Branch-End-Place* is the same end place for all transitions. QPN transition *Branch-Entry* implements the actual branching behavior. It takes a token of color c from *Branch-Begin-Place* and propagates it to a branch behavior i with the probability specified for branch transition i. Note that all branching probabilities need to sum up to 1.0.

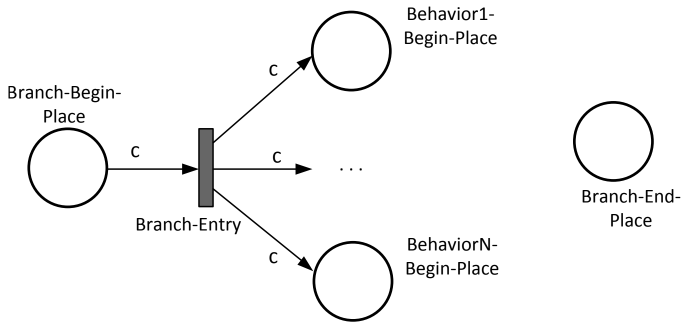


Figure 5.13: QPN Representation of a Branch, cf. Meier (2010)

Formally, a branch with N branch transitions and branching probability pr_i for branch transition i is transformed to a QPN with:

$$P = P \cup \{p_{\textit{Branch-Begin-Place}}, p_{\textit{Branch-End-Place}}, p_{\textit{Behavior1-Begin-Place}}, \ldots, p_{\textit{BehaviorN-Begin-Place}}\}$$
$$T = T \cup \{t_{\textit{Branch-Entry}}\}$$
$$F_C(p_{\textit{Branch-Begin-Place}}) = \{c\}, F_C(p_{\textit{Branch-End-Place}}) = \{c\}$$
$$F_C(p_{\textit{Behavior1-Begin-Place}}) = \{c\}, \ldots, F_C(p_{\textit{BehaviorN-Begin-Place}}) = \{c\}$$
$$F_M(t_{\textit{Branch-Entry}}) = \{m_{t_{\textit{Branch-Entry}},1}, \ldots, m_{t_{\textit{Branch-Entry}},N}\}$$
$$\forall i \in \{1, \ldots, N\} : \ (I^-(p_{\textit{Branch-Begin-Place}}, t_{\textit{Branch-Entry}}))(m_{t_{\textit{Branch-Entry}},i}) = \{c\}_{MS}$$
$$(I^+(p_{\textit{Behavior1-Begin-Place}}, t_{\textit{Branch-Entry}}))(m_{t_{\textit{Branch-Entry}},1}) = \{c\}_{MS},$$
$$\ldots, (I^+(p_{\textit{BehaviorN-Begin-Place}}, t_{\textit{Branch-Entry}}))(m_{t_{\textit{Branch-Entry}},N}) = \{c\}_{MS}$$
$$\forall i \in \{1, \ldots, N\} : \ (W(t_{\textit{Branch-Entry}}))(m_{t_{\textit{Branch-Entry}},i}) = pr\_i$$

Loops

Figure 5.14 shows a loop transformed to a QPN representation. The loop iteration number is specified as PMF where each possible loop iteration count $n_i \in \{n_1, \ldots, n_l\} \subset \mathbb{N}$ is specified with a probability $pr_i \in \{pr_1, \ldots, pr_l\} \subset \mathbb{R}_{\geq 0}$. Thus, there are $l \in \mathbb{N}_0$ different loop iteration counts. The begin and the end of the loop are represented by the places *Loop-Begin-Place* and *Loop-End-Place*.

Transition *Loop-Entry* takes a token of color c and puts a token of color c_i' in place *Loop-Pool* with probability pr_i. Color c_i' represents a loop iteration count of n_i. Transition *Loop-Inner-Entry* then takes a token of color c_i' and puts (i) one token of color c in place *Loop-Body-Begin* as well as (ii) one token of color c_i' in place *Loop-Inner-ColorCode-Place*. The places *Loop-Body-Begin* and *Loop-Body-End* represent the loop body behavior. Place *Loop-Inner-ColorCode-Place* is intended to *save* the color code c_i' while the loop body behavior is processed. Transition *Loop-Inner-Exit* takes a token of color c from place *Loop-Body-End* and a token of color c_i' from place *Loop-Inner-ColorCode-Place*, and puts a token of color c_i' in place *Loop-Depository*. Transition *Loop-Exit* takes a token from place *Loop-Depository* and *decides* based on token color c_i' if the loop body is iterated again (putting a token of color c_i' back to the *Loop-Pool*) or if the loop is terminated (putting a token of color c to place *Loop-End-Place*). The decision can be made based solely on the color code c_i'. Thus, different tokens of the same color code cannot be distinguished. This is why it cannot be guaranteed that the loop is processed *exactly* n_i times. Instead, the loop iteration counts are handled probabilistically. For a token of color code c'_i, the *Loop-Exit* transition terminates the loop with a probability of $1/n_i$ and re-enters the loop with a probability of $1 - (1/n_i)$. The decision whether the loop is re-entered thus behaves like a Bernoulli random variable. Therefore, each loop iteration count n_i is modeled with a geometric distribution with an expected value of n_i.

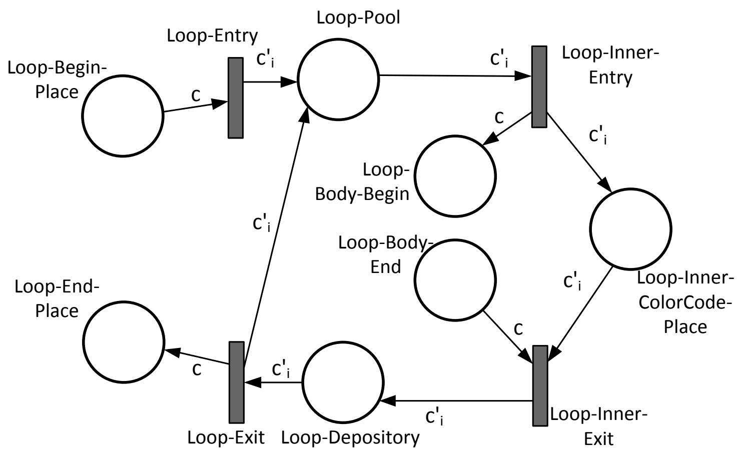
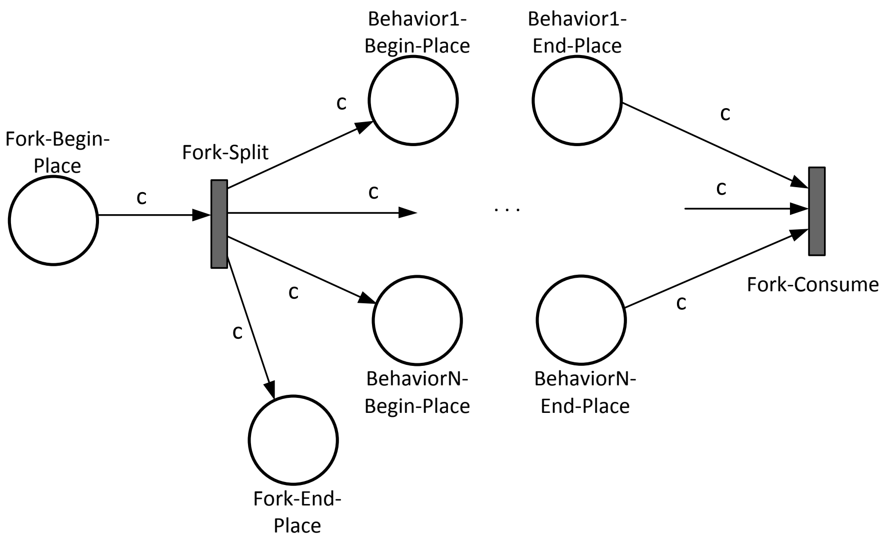


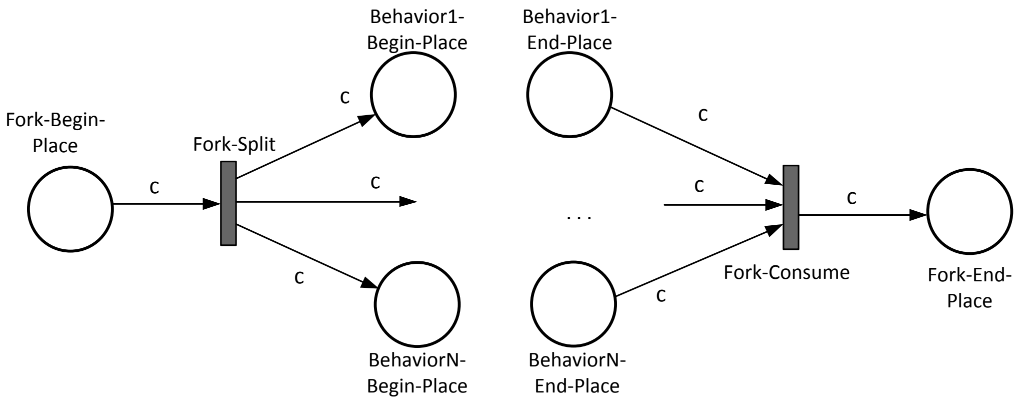
Figure 5.14: QPN Representation of a Loop, cf. Meier (2010)

Formally, a loop with l loop iteration counts $\{n_1, \ldots, n_l\}$ with probabilities $\{pr_1, \ldots, pr_l\}$ is transformed to a QPN with:

$$
\begin{aligned}
&P = P \cup \{p_{\textit{Loop-Begin-Place}}, p_{\textit{Loop-End-Place}}, p_{\textit{Loop-Pool}}, p_{\textit{Loop-Depository}},\\
&\qquad p_{\textit{Loop-Body-Begin}}, p_{\textit{Loop-Body-End}}, p_{\textit{Loop-Inner-ColorCode-Place}}\}\\
&T = T \cup \{t_{\textit{Loop-Entry}}, t_{\textit{Loop-Exit}}, t_{\textit{Loop-Inner-Entry}}, t_{\textit{Loop-Inner-Exit}}\}\\
&F_C(p_{\textit{Loop-Begin-Place}}) = \{c\}, F_C(p_{\textit{Loop-End-Place}}) = \{c\},\\
&F_C(p_{\textit{Loop-Body-Begin}}) = \{c\}, F_C(p_{\textit{Loop-Body-End}}) = \{c\},\\
&F_C(p_{\textit{Loop-Pool}}) = \{c'_1, \ldots, c'_n\},\\
&F_C(p_{\textit{Loop-Depository}}) = \{c'_1, \ldots, c'_l\},\\
&F_C(p_{\textit{Loop-Inner-ColorCode-Place}}) = \{c'_1, \ldots, c'_l\}\\
&F_M(t_{\textit{Loop-Entry}}) = \{m_{t_{\textit{Loop-Entry}},1}, \ldots, m_{t_{\textit{Loop-Entry}},l}\}\\
&F_M(t_{\textit{Loop-Exit}}) = \{m_{t_{\textit{Loop-Exit}},1}, \ldots, m_{t_{\textit{Loop-Exit}},l},\\
&\qquad m_{t_{\textit{Loop-Exit}},l+1}, \ldots, m_{t_{\textit{Loop-Exit}},2l}\}\\
&F_M(t_{\textit{Loop-Inner-Entry}}) = \{m_{t_{\textit{Loop-Inner-Entry}},1}, \ldots, m_{t_{\textit{Loop-Inner-Entry}},l}\}\\
&F_M(t_{\textit{Loop-Inner-Exit}}) = \{m_{t_{\textit{Loop-Inner-Exit}},1}, \ldots, m_{t_{\textit{Loop-Inner-Exit}},l}\}\\
&\forall i \in \{1, \ldots, l\}: \ (I^-(p_{\textit{Loop-Begin-Place}}, t_{\textit{Loop-Entry}}))(m_{t_{\textit{Loop-Entry}},i}) = \{c\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^+(p_{\textit{Loop-Pool}}, t_{\textit{Loop-Entry}}))(m_{t_{\textit{Loop-Entry}},i}) = \{c'_i\}_{MS}\\
&\forall i \in \{1, \ldots, 2l\}: \ (I^-(p_{\textit{Loop-Depository}}, t_{\textit{Loop-Exit}}))(m_{t_{\textit{Loop-Exit}},i}) = \{c'_i\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^+(p_{\textit{Loop-Pool}}, t_{\textit{Loop-Exit}}))(m_{t_{\textit{Loop-Exit}},i}) = \{c'_i\}_{MS}\\
&\forall i \in \{l+1, \ldots, 2l\}: \ (I^+(p_{\textit{Loop-End-Place}}, t_{\textit{Loop-Exit}}))(m_{t_{\textit{Loop-Exit}},i}) = \{c\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^-(p_{\textit{Loop-Pool}}, t_{\textit{Loop-Inner-Entry}}))(m_{t_{\textit{Loop-Inner-Entry}},i}) = \{c'_i\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^+(p_{\textit{Loop-Body-Begin}}, t_{\textit{Loop-Inner-Entry}}))(m_{t_{\textit{Loop-Inner-Entry}},i}) = \{c\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^+(p_{\textit{Loop-Inner-ColorCode-Place}}, t_{\textit{Loop-Inner-Entry}}))(m_{t_{\textit{Loop-Inner-Entry}},i}) = \{c'_i\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^-(p_{\textit{Loop-Body-End}}, t_{\textit{Loop-Inner-Exit}}))(m_{t_{\textit{Loop-Inner-Entry}},i}) = \{c\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^-(p_{\textit{Loop-Inner-ColorCode-Place}}, t_{\textit{Loop-Inner-Exit}}))(m_{t_{\textit{Loop-Inner-Entry}},i}) = \{c'_i\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (I^+(p_{\textit{Loop-Depository}}, t_{\textit{Loop-Inner-Exit}}))(m_{t_{\textit{Loop-Inner-Entry}},i}) = \{c'_i\}_{MS}\\
&\forall i \in \{1, \ldots, l\}: \ (W(t_{\textit{Loop-Entry}}))(m_{t_{\textit{Loop-Entry}},i}) = pr_i\\
&\forall i \in \{1, \ldots, l\}: \ (W(t_{\textit{Loop-Exit}}))(m_{t_{\textit{Loop-Exit}},i}) = 1 - (1/n_i)\\
&\forall i \in \{l+1, \ldots, 2l\}: \ (W(t_{\textit{Loop-Exit}}))(m_{t_{\textit{Loop-Exit}},i}) = 1/n_i
\end{aligned}
$$



(a) QPN Representation of a Fork without Synchronization Barrier



(b) QPN Representation of a Fork with Synchronization Barrier

Figure 5.15: QPN Representation of a Fork, cf. Meier (2010)

Forks

We distinguish forks with and without a synchronization barrier. Figure 5.15 shows a fork of N different behaviors represented as a QPN. The begin and end of the fork are represented by the QPN places *Fork-Begin-Place* and *Fork-End-Place*. The forked behaviors are represented by pairs of places *Behavior1-Begin-Place* and *Behavior1-End-Place* to *BehaviorN-Begin-Place* and *BehaviorN-End-Place*. Transition *Fork-Split* splits the N behaviors by taking a token of color c from the *Fork-Begin-Place* and putting a token of color c in each place *Behavior1-Begin-Place* to *BehaviorN-Begin-Place*. If there is no synchronization barrier, transition *Fork-Split* furthermore propagates a token of color c to *Fork-End-Place*. Transition *Fork-Consume* collects tokens of color c from places *Behavior1-End-Place* to *BehaviorN-End-Place*. In case there is no synchronization barrier, the tokens are not propagated, i.e., they are *destroyed*. In case there is a synchronization barrier, transition *Fork-Consume* propagates a token of color c to the *Fork-End-Place*.

Formally, a fork of N behaviors is transformed to a QPN with:

$$
\begin{aligned}
&P = P \cup \{p_{\textit{Fork-Begin-Place}}, p_{\textit{Fork-End-Place}},\\
&\qquad p_{\textit{Behavior1-Begin-Place}}, \dots, p_{\textit{BehaviorN-Begin-Place}},\\
&\qquad p_{\textit{Behavior1-End-Place}}, \dots, p_{\textit{BehaviorN-End-Place}}\}\\
&T = T \cup \{t_{\textit{Fork-Split}}, t_{\textit{Fork-Consume}}\}\\
&F_C(p_{\textit{Fork-Begin-Place}}) = \{c\}, F_C(p_{\textit{Fork-End-Place}}) = \{c\}\\
&F_C(p_{\textit{Behavior1-Begin-Place}}) = \{c\}, \dots, F_C(p_{\textit{BehaviorN-Begin-Place}}) = \{c\}\\
&F_C(p_{\textit{Behavior1-End-Place}}) = \{c\}, \dots, F_C(p_{\textit{BehaviorN-End-Place}}) = \{c\}\\
&F_M(t_{\textit{Fork-Split}}) = \{m_{t_{\textit{Fork-Split}},1}\}\\
&F_M(t_{\textit{Fork-Consume}}) = \{m_{t_{\textit{Fork-Consume}},1}\}\\
&(I^-(p_{\textit{Fork-Begin-Place}}, t_{\textit{Fork-Split}}))(m_{t_{\textit{Fork-Split}},1}) = \{c\}_{MS}\\
&(I^+(p_{\textit{Behavior1-Begin-Place}}, t_{\textit{Fork-Split}}))(m_{t_{\textit{Fork-Split}},1}) = \{c\}_{MS},\\
&\quad \dots, (I^+(p_{\textit{BehaviorN-Begin-Place}}, t_{\textit{Fork-Split}}))(m_{t_{\textit{Fork-Split}},1}) = \{c\}_{MS}\\
&(I^-(p_{\textit{Behavior1-End-Place}}, t_{\textit{Fork-Consume}}))(m_{t_{\textit{Fork-Consume}},1}) = \{c\}_{MS},\\
&\quad \dots, (I^-(p_{\textit{BehaviorN-End-Place}}, t_{\textit{Fork-Consume}}))(m_{t_{\textit{Fork-Consume}},1}) = \{c\}_{MS}
\end{aligned}
$$

If there is a synchronization barrier, the following forward incidence function needs to be added:

$$(I^+(p_{\textit{Fork-End-Place}}, t_{\textit{Fork-Consume}}))(m_{t_{\textit{Fork-Consume}},1}) = \{c\}_{MS}.$$

If there is no synchronization barrier, the following forward incidence function needs to be added:

$$(I^+(p_{\textit{Fork-End-Place}}, t_{\textit{Fork-Split}}))(m_{t_{\textit{Fork-Split}},1}) = \{c\}_{MS}.$$

Acquire/Release Passive Resources

Figure 5.16 illustrates how one can transform AcquireActions and ReleaseActions to QPNs. The corresponding PassiveResource is modeled as tokens of new color p that are deposited in the *Passive-Resource-Place*. The capacity of the PassiveResource is mapped to the initial number of tokens of color p in place *Passive-Resource-Place*. An AcquireAction is represented by two places *Acquire-Begin-Place* and *Acquire-End-Place*. Transition *Acquire-Resource* requires a token of color p from place *Passive-Resource-Place* to propagate a token of color c from *Acquire-Begin-Place* to *Acquire-End-Place*. A ReleaseAction in turn is represented by two places *Release-Begin-Place* and *Release-End-Place*. Transition *Release-Resource* takes a token of color c from the *Release-Begin-Place* and puts both a token of color c in the *Release-End-Place* as well as a token of color p back in the *Passive-Resource-Place*.

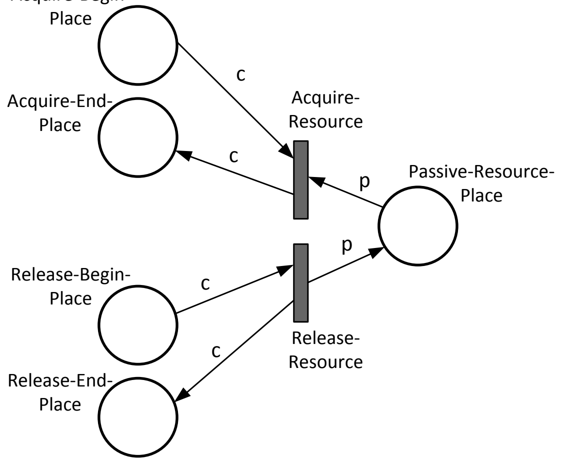


Figure 5.16: QPN Representation of Acquire/Release Actions, cf. Meier (2010)

Formally, a pair of Acquire- and ReleaseActions is transformed to a QPN with:

$$
\begin{aligned}
&P = P \cup \{p_{\textit{Acquire-Begin-Place}}, p_{\textit{Acquire-End-Place}}, p_{\textit{Release-Begin-Place}}, p_{\textit{Release-End-Place}}\\
&\qquad p_{\textit{Passive-Resource-Place}}\}\\
&T = T \cup \{t_{\textit{Acquire-Resource}}, t_{\textit{Release-Resource}}\}\\
&C = C \cup \{p\}\\
&F_C(p_{\textit{Acquire-Begin-Place}}) = \{c\}, F_C(p_{\textit{Acquire-End-Place}}) = \{c\}\\
&F_C(p_{\textit{Release-Begin-Place}}) = \{c\}, F_C(p_{\textit{Release-End-Place}}) = \{c\}\\
&F_C(p_{\textit{Passive-Resource-Place}}) = \{p\}\\
&F_M(t_{\textit{Acquire-Resource}}) = \{m_{t_{\textit{Acquire-Resource}},1}\}\\
&F_M(t_{\textit{Release-Resource}}) = \{m_{t_{\textit{Release-Resource}},1}\}\\
&F_C^0(p_{\textit{Passive-Resource-Place}}) = \{ \underbrace{p, \ldots, p}_{\text{capacity of passive resource}} \}_{MS}\\
&(I^-(p_{\textit{Acquire-Begin-Place}}, t_{\textit{Acquire-Resource}}))(m_{t_{\textit{Acquire-Resource}},1}) = \{c\}_{MS}\\
&(I^-(p_{\textit{Passive-Resource-Place}}, t_{\textit{Acquire-Resource}}))(m_{t_{\textit{Acquire-Resource}},1}) = \{p\}_{MS}\\
&(I^+(p_{\textit{Acquire-End-Place}}, t_{\textit{Acquire-Resource}}))(m_{t_{\textit{Acquire-Resource}},1}) = \{c\}_{MS}\\
&(I^-(p_{\textit{Release-Begin-Place}}, t_{\textit{Release-Resource}}))(m_{t_{\textit{Release-Resource}},1}) = \{c\}_{MS}\\
&(I^+(p_{\textit{Passive-Resource-Place}}, t_{\textit{Release-Resource}}))(m_{t_{\textit{Release-Resource}},1}) = \{p\}_{MS}\\
&(I^+(p_{\textit{Release-End-Place}}, t_{\textit{Release-Resource}}))(m_{t_{\textit{Release-Resource}},1}) = \{c\}_{MS}
\end{aligned}
$$

Response Times

The model entity ResponseTime is translated to a QPN as shown in Figure 5.17. It is represented by the two places *ResponseTime-Begin-Place* and *ResponseTime-End-Place*. Transitions *RT-Entry* and *RT-Exit* propagate the response time delay to the immediate queueing place *Delay-Place* via tokens of new color r. The service demand of tokens of color r is set to the specified response time value, given as a RandomVariable.

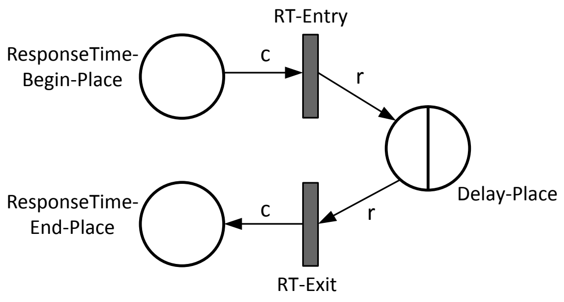


Figure 5.17: QPN Representation of Model Entity ResponseTime

Formally, a ResponseTime characterized with value *response_time* of type RandomVariable is transformed to a QPN with:

$$
\begin{aligned}
&P = P \cup \{p_{\textit{ResponseTime-Begin-Place}}, p_{\textit{ResponseTime-End-Place}}, p_{\textit{Delay-Place}}\}\\
&T = T \cup \{t_{\textit{RT-Entry}}, t_{\textit{RT-Exit}}\}\\
&C = C \cup \{r\}\\
&Q = Q \cup \{q\}, \tilde{Q}_2 = \tilde{Q}_2 \cup \{q\}\\
&F_C(p_{\textit{ResponseTime-Begin-Place}}) = \{c\}, F_C(p_{\textit{ResponseTime-End-Place}}) = \{c\}, F_C(p_{\textit{Delay-Place}}) = \{r\}\\
&F_M(t_{\textit{RT-Entry}}) = \{m_{t_{\textit{RT-Entry}},1}\}\\
&F_M(t_{\textit{RT-Exit}}) = \{m_{t_{\textit{RT-Exit}},1}\}\\
&(I^-(p_{\textit{ResponseTime-Begin-Place}}, t_{\textit{RT-Entry}}))(m_{t_{\textit{RT-Entry}},1}) = \{c\}_{MS}\\
&(I^+(p_{\textit{Delay-Place}}, t_{\textit{RT-Entry}}))(m_{t_{\textit{RT-Entry}},1}) = \{r\}_{MS}\\
&(I^-(p_{\textit{Delay-Place}}, t_{\textit{RT-Exit}}))(m_{t_{\textit{RT-Exit}},1}) = \{r\}_{MS}\\
&(I^+(p_{\textit{ResponseTime-End-Place}}, t_{\textit{RT-Exit}}))(m_{t_{\textit{RT-Exit}},1}) = \{c\}_{MS}\\
&\tilde{Q}_P(p_{\textit{Delay-Place}}) = \{q\}\\
&(D_Q(q))(demand(r)) = response\_time
\end{aligned}
$$

Resource Demands

The model entity ResourceDemand is translated to a QPN as shown in Figure 5.18. It is represented by the two places *ResourceDemand-Begin-Place* and *ResourceDemand-End-Place*. Transitions *RD-Entry* and *RD-Exit* propagate the demand via token of new color r to a QPN subnet representing the target resource type of the Container where the ResourceDemand is issued.

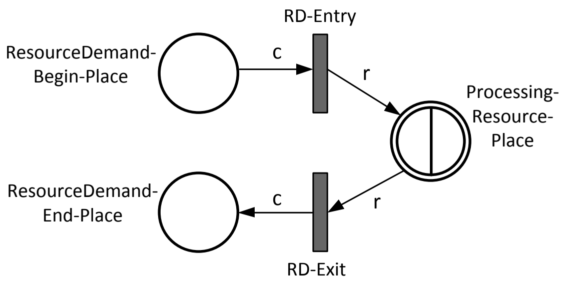


Figure 5.18: QPN Representation of Model Entity ResourceDemand

In the following QPN formalization, the processing resource is assumed to be mapped to a queueing place with description *processing_resource_description*. Then, a ResourceDemand characterized with value *resource_demand* of type RandomVariable is transformed to a QPN with:

$$\begin{aligned}
&P = P \cup \{p_{\textit{ResourceDemand-Begin-Place}}, p_{\textit{ResourceDemand-End-Place}}, p_{\textit{Processing-Resource-Place}}\}\\
&T = T \cup \{t_{\textit{RD-Entry}}, t_{\textit{RD-Exit}}\}\\
&C = C \cup \{r\}\\
&Q = Q \cup \{prq\}, \tilde{Q}_1 = \tilde{Q}_1 \cup \{prq\}\\
&F_C(p_{\textit{ResourceDemand-Begin-Place}}) = \{c\}, F_C(p_{\textit{ResourceDemand-End-Place}}) = \{c\}\\
&F_C(p_{\textit{Processing-Resource-Place}}) = \{r\}\\
&F_M(t_{\textit{RD-Entry}}) = \{m_{t_{\textit{RD-Entry}},1}\}\\
&F_M(t_{\textit{RD-Exit}}) = \{m_{t_{\textit{RD-Exit}},1}\}\\
&(I^-(p_{\textit{ResourceDemand-Begin-Place}}, t_{\textit{RD-Entry}}))(m_{t_{\textit{RD-Entry}},1}) = \{c\}_{MS}\\
&(I^+(p_{\textit{Processing-Resource-Place}}, t_{\textit{RD-Entry}}))(m_{t_{\textit{RD-Entry}},1}) = \{r\}_{MS}\\
&(I^-(p_{\textit{Processing-Resource-Place}}, t_{\textit{RD-Exit}}))(m_{t_{\textit{RD-Exit}},1}) = \{r\}_{MS}\\
&(I^+(p_{\textit{ResourceDemand-End-Place}}, t_{\textit{RD-Exit}}))(m_{t_{\textit{RD-Exit}},1}) = \{c\}_{MS}\\
&\tilde{Q}_P(p_{\textit{Processing-Resource-Place}}) = \{prq\}\\
&D_Q(prq) = \text{processing\_resource\_description}\\
&(D_Q(prq))(demand(r)) = resource\_demand
\end{aligned}$$

5.3.2 Transformation to Layered Queueing Networks

As in the transformation to QPNs, for the transformation from DML instances to LQNs the following abstract modeling constructs need to be considered.

- Open workload (model entity OpenWorkloadType)
- Closed workload (model entity ClosedWorkloadType)
- Calls (model entities SystemCallUserAction, ExternalCall)
- Branches (model entities BranchUserAction, BranchAction)
- Loops (model entities LoopUserAction, LoopAction, ExternalCallFrequency)
- Forks with and without synchronization barrier (model entity ForkAction)

- Acquire and release passive resource (model entities AcquireAction and ReleaseAction)
- Response Times (model entity ResponseTime and DelayUserAction)
- Resource Demands (model entity ResourceDemand)

Koziolek (2008) describes a transformation from PCM instances to LQN. Since PCM also contains modeling constructs for calls, branches, loops, and so on, we can make use of the developed transformation concepts. The above-mentioned modeling constructs are transformed to the LQN concepts of tasks and processors (see Section 2.2) briefly described as follows: Each service behavior model is mapped to a LQN task. The control flow behavior can be mapped to the corresponding counterparts in LQN task activity graphs. Service calls are realized by calling other LQN task entries. The tasks that are created for service behavior models run on dummy processors. Service demands are sent to separate LQN tasks with processors that represent the target resources (e.g., application server CPU). Such LQN processors are defined by M/M/n queues, i.e., limited to exponentially distributed service times. Moreover, the inter-arrival times of open workloads are also limited to exponential distributions. For details of the transformation, we refer the reader to Koziolek (2008).

5.3.3 Bounds Analysis

In this section, we describe how we conduct an asymptotic bounds analysis based on the modeling abstractions presented in Chapter 4. An asymptotic bounds analysis is a technique to calculate performance bounds such as lowest average response time and highest possible throughput for an underlying system. Bounds analysis is fast because it works with several approximations that cause inaccuracies, however, it can be good enough to make quick decisions, when approximative performance results are sufficient (Menascé et al., 1994; Bolch et al., 1998; Menascé et al., 2004b). As shown in Figure 5.9, the input of the bounds analysis is the architecture-level performance model and the callstack model, which is the output of the pre-processing step described in Section 5.2.

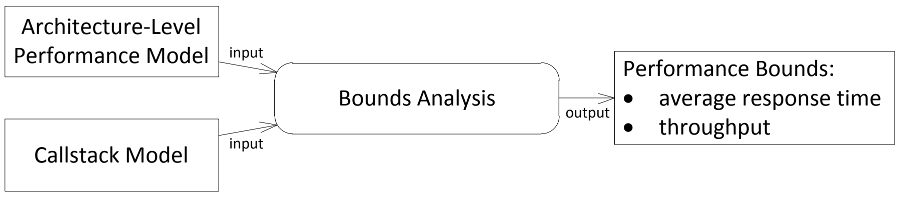


Figure 5.19: Input and Output of Bounds Analysis

The analysis starts with the usage profile model (see Section 4.3) and then traverses the application architecture model. A bounds analysis is conducted per UsageScenario, i.e., multiple UsageScenarios are not considered. Thus, the lowest average response time as well as the highest possible throughput are calculated for one UsageScenario. Response times of individual service calls cannot be considered.

Starting with a UsageScenario, the corresponding ServiceBehavior is traversed. Whenever the traversal reaches a SystemCallUserAction, the callstack of stack frames is used to navigate to the corresponding ServiceBehaviorAbstraction. Whenever an ExternalCall is reached, the stack frames are used to navigate to the next ServiceBehaviorAbstraction. Whenever a ModelVariable is accessed, it is read from the current stack frame. The traversal for a UsageScenario *USc* has the following two goals:

1. Identification of the set of different resources $\overline{Res}$ that are stressed by *USc*,

$$\overline{Res} = \{Res_0, Res_1, \ldots, Res_n\}.$$

We denote Res_0 as the delay resource, i.e., where all DelayUserActions and ResponseTimes are scheduled.

2. Calculation of the mean resource demands $\overline{D}$ for the n identified resources for one traversal of *USc*,
$$\overline{D} = \{D_0, \ldots, D_n\}.$$
More specifically, we distinguish between resource demands that are issued in an asynchronous context, i.e., as part of a ForkAction, or in a synchronous context. Hence, we distinguish $\overline{D^{async}}$ and $\overline{D^{sync}}$ with $\overline{D} = \overline{D^{async}} + \overline{D^{sync}}$.

When traversing the performance model, the following modeling constructs are considered:

- Branches (model entities BranchUserAction, BranchAction),
- Loops (model entities LoopUserAction, LoopAction, ExternalCallFrequency),
- Calls (model entities SystemCallUserAction, ExternalCall),
- Forks without a synchronization barrier (model entity ForkAction),
- Resource Demands (model entity ResourceDemand),
- Response Times (model entity ResponseTime and model entity DelayUserAction).

Note that AcquireActions, respectively ReleaseActions, are ignored. The asymptotic bounds analysis neglects passive resources. For the same reason, forks with a synchronization barrier are ignored, too.

We now describe how $\overline{Res}$ and $\overline{D}$ are derived. We navigate the tree of behavior abstractions using a depth first traversal. Algorithm 6 shows the traversal in pseudo-code. We choose an iterative pre-order depth-first approach (Cormen et al., 2009) with a helper stack (not to be confused with the callstack model). During the traversal, we keep track of (i) the current behavior abstraction denoted with variable *behavior*, (ii) the mean number of visits of the current behavior denoted with variable *visits*, and (iii) a flag indicating whether the current behavior is processed as part of an asynchronous fork or not, denoted with variable *isAsync*.

The algorithm is initialized with $\overline{D} = \emptyset$ and $\overline{Res} = \emptyset$, and invoked with the method call $ComputeOverline{D}And\overline{Res}(startbehavior, 1.0, FALSE)$. The method iterates through the actions of a behavior abstraction and handles the individual actions as follows:

- When a branch is reached, each branch transition is traversed. The mean number of visits of each branch transitions is calculated taking the branching probability into account.
- When a loop is reached, the loop body is traversed. The mean number of visits of the loop body is calculated using the given probabilities for the loop iteration numbers.
- When a call is reached, the target behavior of the call is traversed.
- When a fork without synchronization barrier is reached, each forked behavior is traversed, marking the behavior as asynchronously processed.
- When a resource demand is reached, on the one hand the target resource Res_i is resolved. The target resource is obtained dependent on (i) the specified ResourceType of the resource demand and (ii) the Container where the current behavior is deployed on. On the other hand, the mean resource demand D_i is updated. The product of the expected value of the given random variable and the mean number of visits of the current behavior is added to D_i. Depending on flag *isAsync*, either D_i^{async} or D_i^{sync} is set.

```
$\mathit{Compute}\overline{D}\mathit{And}\overline{Res}(\mathit{behavior}, \mathit{visits}, \mathit{isAsync})$
begin
    helperStack $\leftarrow$ empty stack
    while not helperStack.isEmpty() or $\mathit{behavior}$ <> $\mathit{NULL}$ do
        foreach action $a$ in $\mathit{behavior}$ do
            if $a$ is Branch with branch transitions $\{t_1, \ldots, t_N\}$
              with probabilities $\{pr_1, \ldots, pr_N\}$ then
                for $i \leftarrow 1$ to $N$ do
                    helperStack.push( $\mathit{Tuple}(t_i, \mathit{visits} * pr_i, \mathit{isAsync})$ )
                end
            if $a$ is Loop with loop body $b$ and loop iteration numbers $\{n_1, \ldots, n_l\}$
              with probabilities $\{pr_1, \ldots, pr_l\}$ then
                helperStack.push( $\mathit{Tuple}(b, \sum_{k=1}^{l}(n_k * pr_k), \mathit{isAsync})$ )
            if $a$ is Call with target behavior $t$ then
                helperStack.push( $\mathit{Tuple}(t, \mathit{visits}, \mathit{isAsync})$ )
            if $a$ is Fork with forked behaviors $\{f_1, \ldots, f_N\}$
              without synchronization barrier then
                for $i \leftarrow 1$ to $N$ do
                    helperStack.push( $\mathit{Tuple}(f_i, \mathit{visits}, \mathit{TRUE})$ )
                end
            if $a$ is ResourceDemand with value $v$ and resource type $t$ then
                $\mathit{helper} \leftarrow$ resource of type $t$ in container $\mathit{deploymentContainer}(a)$
                $\overline{Res} \leftarrow \overline{Res} \cup \{\mathit{helper}\}$
                $i \leftarrow$ index of $\mathit{helper}$ in $\overline{Res}$
                if $\mathit{isAsync}$ then $D_i^{async} \leftarrow D_i^{async} + \mathit{visits} * \mathit{mean}(v)$
                else $D_i^{sync} \leftarrow D_i^{sync} + \mathit{visits} * \mathit{mean}(v)$
            if $a$ is ResponseTime with value $v$ then
                if $\mathit{isAsync}$ then $D_0^{async} \leftarrow D_0^{async} + \mathit{visits} * \mathit{mean}(v)$
                else $D_0^{sync} \leftarrow D_0^{sync} + \mathit{visits} * \mathit{mean}(v)$
        end
        $\mathit{Tuple}(\mathit{behavior}, \mathit{visits}, \mathit{isAsync}) \leftarrow$ helperStack.pop()
    end
end
```

Algorithm 6: Compute Mean Resource Demands $\overline{D}$ for Resources $\overline{Res}$

- When a response time is reached, D_0 is set accordingly as a resource demand for delay resource Res_0.

The algorithm terminates because cycles of service behaviors are not allowed (see Section 4.1.3). Having sets $\overline{Res}$ and $\overline{D}$ identified, we now describe how we apply asymptotic bounds analysis. We first describe how we obtain an upper bound for the throughput $\overline{X}$ of the system and then explain how a lower bound for the average response time $\overline{R}$ of the considered UsageScenario can be derived. We extend the approaches presented in Menascé et al. (1994); Bolch et al. (1998); Menascé et al. (2004b) since we differentiate between D^{async} and D^{sync}. The bounding behavior is determined by the bottleneck resource, i.e., the resource with the highest utilization.

Using the Utilization Law $U_i = D_i * X_i$ where X_i is the throughput, D_i is the mean resource demand and U_i is the utilization at resource Res_i, we obtain

$$X_i = \frac{U_i}{D_i} \leq \frac{1}{D_i}.$$

An upper asymptotic bound for $\overline{X}$ is thus

$$\overline{X} \leq min_{0 \leq i \leq n} \left\{ \frac{1}{D_i} \right\} = min_{1 \leq i \leq n} \left\{ \frac{1}{D_i} \right\}.$$

Using Little's Law (Little, 1961) and $\overline{R} \geq \sum_{i=0}^{n} D_i^{sync}$, we obtain

$$N = \overline{R} * \overline{X} \geq \left(\sum_{i=0}^{n} D_i^{sync} \right) * \overline{X} \Longleftrightarrow \overline{X} \leq \frac{N}{\sum_{i=0}^{n} D_i^{sync}},$$

where N is the number of concurrent users in the system. The number of users is limited by the number of clients in case of a closed workload. In case of an open workload N is unknown. As upper asymptotic bounds for the throughput we formulate

$$\overline{X} \leq min \left\{ \frac{N}{\sum_{i=0}^{n} D_i^{sync}}, min_{1 \leq i \leq n} \left\{ \frac{1}{D_i} \right\} \right\}.$$

These bounds are then used together with Little's Law to derive a lower asymptotic bound for the average response time $\overline{R}$, if N is known:

$$\overline{R} = \frac{N}{\overline{X}} \geq max \left\{ \sum_{i=0}^{n} D_i^{sync}, N * max_{1 \leq i \leq n} \{D_i\} \right\}.$$

Furthermore, for an open workload with an arrival rate λ of usage scenario *USc*, if $\lambda < \overline{X}$ then we can estimate the utilization of the resources using the utilization law with

$$U_j = D_j * \lambda.$$

The asymptotic bounds analysis cannot consider software bottlenecks as well as load-dependent resource demands. However, it offers a quick approximation of the lowest average response time and highest possible throughput for a given UsageScenario.

5.4 Tailoring

The prediction process is tailored to a performance query (see Figure 5.1). A performance query specifies which performance metrics of which entities are of interest for the performance prediction. For instance, when triggering a performance prediction, the user may ask for the average response time of a specific service or for the utilization of a specific resource such as the database server. Furthermore, given that the focus is on performance prediction techniques applied during system operation, the speed of the performance prediction itself, i.e., the time-to-result of a performance prediction, is of importance. In case the performance prediction is intended to be used to find a system configuration that provides a certain level of service performance and resource efficiency for an upcoming workload scenario, the results of the performance prediction have to be available *before* the expected workload scenario stresses the system. The prediction process we propose takes these time constraints into account. We provide the option to trade-off between prediction speed and prediction accuracy. In situations where the prediction speed is critical, the prediction process provides the option to speed up the prediction, however, this speed up comes at the cost of prediction accuracy. To sum up, the prediction process is tailored to the requested performance metrics as well as to a given specification of how to trade-off between prediction accuracy and time-to-result (Thereska et al., 2005; Kounev et al., 2010). Both the demanded performance metrics and the trade-off specification form the performance query that triggers the performance prediction.

Section 5.4.1 describes the supported performance metrics. Section 5.4.2 formalizes how the trade-off between prediction accuracy and time-to-result can be specified. Section 5.4.3 describes the degrees-of-freedom of the prediction process, i.e., its configurations options. Section 5.4.4 and Section 5.4.5 describe the tailoring mechanisms for the model composition step and the model solving step, respectively.

5.4.1 Performance Metrics

Performance metrics are considered for two types of performance-relevant model entities. We distinguish metrics for service calls and metrics for resources. For service calls, we consider end-user metric types such as response time, i.e., elapsed time between request and response, and throughput. For resources, we consider their utilization. We furthermore provide different characterizations of the response time distribution. We characterize the resulting response time distribution by a sample set, by its percentiles or by its mean value.

Location	Metric	Aggregation
Service Calls	Response Time	Sample set
		Percentile
		Average
	Throughput	Average
Resources	Throughput	Average
	Utilization	Average

Table 5.2: Performance Metrics And Aggregations Considered for Tailoring

The model solving approach highly depends on the level of detail of the requested performance metrics. For instance, predicting average response times does not require the same abstraction level as providing a representative sample set to characterize a response time distribution. Table 5.2 lists the metrics and aggregations that are supported by the tailored prediction process. A set of demanded metrics can be formalized as $DM = \{dm_1, \ldots, dm_n\}$

where dm_i is a tuple (l_i, m_i, a_i) with l_i denoting a location, m_i a metric name and a_i an aggregation as shown in Table 5.2.

5.4.2 Trade-Off Between Prediction Accuracy and Time-To-Result

As mentioned previously, in situations where the prediction speed is critical, the prediction process provides the option to speed up the prediction at the expense of prediction accuracy. Note that in this context, *prediction accuracy* refers to the accuracy of the model solving approach, i.e., *not* to the representativeness of the considered models' themselves. It is not intended to specify real-time constraints for the prediction process but to allow specifying how to trade-off between prediction accuracy and time-to-result.

There are generic trade-off weights that are ranked in an ordinal scale, they are defined as an ordered set $W = \{w_\perp := w_1, \ldots, w_K =: w_\top\}$. Weight $w_\top$ has the semantics of fastest prediction speed compared to the other weights $w \in W$. Weight $w_\perp$ has the semantics of highest prediction accuracy compared to the other weights $w \in W$. A trade-off specification is then given by a selected weight $dw \in W$ that is chosen by the issuer of the performance query.

Based on the trade-off specification, i.e., a selected trade-off weight dw, the prediction process is tailored within its degrees-of-freedom that are described in the next section.

5.4.3 Degrees-of-Freedom

The performance prediction process is triggered by a performance query. It is tailored to a set of demanded performance metrics DM as well as to a given specification dw of how to trade-off between prediction accuracy and time-to-result. The underlying performance model is an architecture-level performance model as described in Chapter 4. This section provides a general overview of the degrees-of-freedom of the prediction process.

As shown in Figure 5.1, both of the steps model composition and model solving are tailored. In the model composition step, it is decided which parts of the source model, i.e., which parts of the architecture-level performance model, are considered for the performance prediction. In the model solving step, it is decided which model solving technique is applied. Furthermore, each model solving technique itself comes with its own degrees-of-freedom. Hence, the applied model solving technique itself can also be tailored to the performance query.

Given DM and dw, the model composition and model solving steps have to be configured to return results for the demanded metrics taking the trade-off specification into account. Thus, for each configuration, it is important to understand how it affects the performance prediction process in terms of predictable performance metrics, prediction accuracy, and prediction speed.

- A fine-grained simulation can provide the best prediction accuracy but at the cost of lowest prediction speed. Complex performance metrics such as response time distributions can be provided. In case only mean value metrics are demanded, the simulation can abstract from complex control flow constructs such as branches or loops.
- Analytical model solvers typically are of lower prediction overhead compared to simulation, but they are often restricted in terms of predictable performance metrics and model input parameters. Analytical solvers such as LQNS (Franks et al., 1996) are often restricted to exponentially distributed resource demands, delays and inter-arrival times (Balsamo et al., 2004), or have limited capabilities to analyze blocking behavior or simultaneous resource possession (Menasce and Virgilio, 2000; Woodside et al., 2006; Gilmore et al., 2005).

The goal of the tailored prediction process is to find a balance between prediction accuracy and overhead. The next two sections (Section 5.4.4 and Section 5.4.5) describe how model composition and model solving can be tailored, using the model abstractions of Chapter 4 as a source model and the solving techniques of Section 5.3.

5.4.4 Tailored Model Composition

As described in Section 5.1, the usage profile model's ScenarioBehaviors determine those parts of the architecture-level performance model that have to be considered for a performance prediction. However, using the model abstractions described in Section 4.1.3), one service may be described by up to three behavior abstraction levels FineGrainedBehavior, CoarseGrainedBehavior and BlackBoxBehavior. This ambiguity is resolved in the tailored model composition step.

A FineGrainedBehavior model provides information about component-internal, performance-relevant control flow including its resource consumption. Loops, branches, forks with or without synchronization barriers, as well as blocking behavior with semaphore semantics can be described. Hence, complex response time distributions can be modeled in a representative manner. A CoarseGrainedBehavior model also provides information about resource demands and the frequency of external service calls, however, the internal service control flow is not modeled. Thus, response time distributions are approximated, but there is no loss in accuracy if response time averages or average resource utilization are to be predicted. A BlackBoxBehavior model provides information about service response times but no information about resource demands. Thus, information about resource utilization cannot be derived.

In the following, we describe how the selection of an appropriate service behavior model is done. The performance query is given as a set of demanded metrics DM and a trade-off specification dw. The selection consists of three steps: *initialization*, *weighting*, and *truncation*. For the description of the three steps, we introduce two helper functions. Assuming each service has a marked service behavior, for a service s we define

$$\mathit{calledServices}(s) := \text{set of services that are directly or indirectly triggered by } s,$$

where the called services are derived from the marked service behaviors via model element ExternalCall. Based on $\mathit{calledServices}(s)$, we also define

$$\mathit{resources}(s) := \bigcup_{s' \in \mathit{calledServices}(s)} \text{set of passive and active resources stressed by } s',$$

i.e, the set of passive and active resources that is stressed when service s and its subsequent services are processed. The three steps are described in the following:

- *Initialization.* Using the ordering FineGrainedBehavior > CoarseGrainedBehavior > BlackBoxBehavior, for each service, the service behavior marking is set to the most fine-grained available service behavior. For instance, for a service with two available service behaviors FineGrainedBehavior and CoarseGrainedBehavior, FineGrainedBehavior is marked.
- *Weighting.* The trade-off specification $dw = w_i \in W = \{w_\perp = w_1, \ldots, w_K = w_\top\}$ determines the target service behavior level. Mapping function $w_i \mapsto ((i - 1) \text{ div } \lceil K/3 \rceil)$ maps $dw = w_i$ to the levels fine-grained (= 0), coarse-grained (= 1) and black-box (= 2). If the target level is fine-grained, we proceed directly with the *truncation* step. If the target level is coarse-grained, for all services where both behavior descriptions FineGrainedBehavior and CoarseGrainedBehavior are available, CoarseGrainedBehavior is marked. If the target level is black box, the following applies:

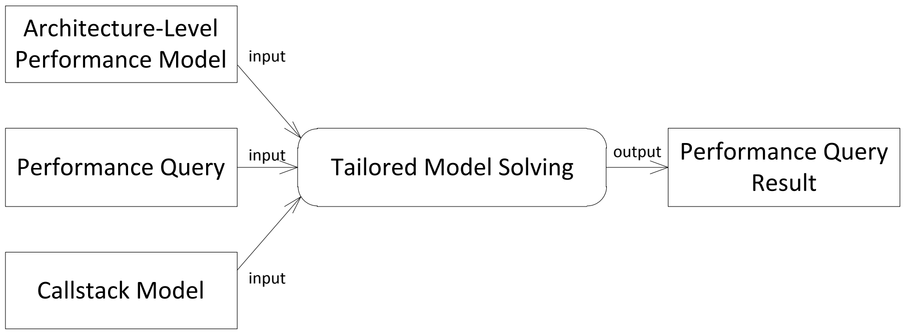


Figure 5.20: Input and Output of Model Solving

  - For all services s where a BlackBoxBehavior description is available, BlackBoxBehavior is marked (i) if $\forall s' \in \mathit{calledServices}(s) : \neg\exists dm_i = (l_i, m_i, a_i) \in \mathit{DM} : s' = l_i$, i.e., if the service call path starting with s does not contain a service for which a metric is requested in *DM*, and (ii) if $\forall r' \in \mathit{resources}(s) : \neg\exists dm_i = (l_i, m_i, a_i) \in \mathit{DM} : r' = l_i$, i.e., if s does not stress a resource for which a metric is requested in *DM*.
  - For all services s where only a FineGrainedBehavior and a CoarseGrainedBehavior is available, CoarseGrainedBehavior is marked.

- *Truncation.* System calls within a UsageScenario that do not involve services or resources that affect a demanded metric can be truncated in the performance prediction. Such calls do not contribute to the result of the performance query and can therefore be omitted. Let S be the set of called system services of the UsageScenarios of the UsageProfileModel. We partition S in two sets $S' := \{s \in S : \forall s' \in \mathit{calledServices}(s) : \neg\exists dm_i = (l_i, m_i, a_i) \in \mathit{DM} : s' = l_i\}$ and $S_{\mathit{DM}} := S \setminus S'$, i.e., S' denotes the set of services called by a UsageScenario whose call paths do not include a demanded metric and S_{DM} denotes the set of services called by a UsageScenario whose call paths *do* include a demanded metric. Each $s' \in S'$ where expressions $\forall r' \in \mathit{resources}(s') : \neg\exists dm_i = (l_i, m_i, a_i) \in \mathit{DM} : r' = l_i$ and $\forall s_{\mathit{DM}} \in S_{\mathit{DM}} : \mathit{resources}(s') \cap \mathit{resources}(s_{\mathit{DM}}) = \emptyset$ hold can then be truncated.

The complexity of the initialization step is $O(n)$ where n is the number of services in the given architecture-level performance model, the complexity of the weighting step is $O(n * m + n * r)$ where m is the number of demanded metrics *DM* and r is the number of resources in the given architecture-level performance model. The truncation step also has a complexity of $O(n * m + n * r + n * r) = O(n * m + n * r)$.

The output of the model composition step is a mark model indicating which service behavior models should be considered for a performance prediction tailored to the given performance query. The decision which service behavior abstraction level is marked depends on the given trade-off specification *dw* and considers the set of demanded metrics *DM*.

5.4.5 Tailored Model Solving

Figure 5.20 shows the input and output of the model solving step. As input, there is the architecture-level performance model, a performance query consisting of demanded metrics *DM* and a trade-off specification *dw* (see Section 5.4.1 and Section 5.4.2) as well as a callstack model as described in Section 5.2. The output of the model solving step are the predicted results for the demanded metrics *DM*. While Section 5.3 describes different model solving techniques, in this section the focus is on *tailoring* the solving techniques to the given performance query. On the one hand, it is decided which of the available model solving techniques is appropriate for the performance query. On the other hand, each model solving technique itself comes with its own configuration options and thus itself can be tailored to the query. The following tailored process is based on three model solving techniques, namely: (i) bounds analysis (Section 5.3.3), (ii) transformation to LQN (Section 5.3.2) where the resulting LQN is solved with the analytical solver LQNS (Franks et al., 2011; Franks, 1999), and (iii) transformation to QPN (Section 5.3.1) where the resulting QPN is solved by simulation with SimQPN (Kounev and Buchmann, 2006; Spinner et al., 2012).

As described in Section 5.3.3, a bounds analysis provides quick asymptotic bounds for average throughput and average response time, but comes at the cost of prediction accuracy. Furthermore, in case of an open workload, a utilization prediction can be quickly derived. Thus, let *UsgSc1* be the only instance of UsageScenario in the given usage profile model, the criteria to decide between (i) and (ii), (iii) are then given by:

- If $dw = w_\top$, and *UsgSc1* has a closed workload, and only the average response time and/or throughput of *UsgSc1* is requested, choose bounds analysis.
- If $dw = w_\top$, and *UsgSc1* has an open workload, and only the average response time and/or throughput of *UsgSc1* and/or resource utilization is requested, choose bounds analysis.
- If *UsgSc1* has an open workload, and only resource utilization is requested, choose bounds analysis.
- In all other cases proceed with the other model solving techniques (ii), (iii).

As described in Section 5.3.2, LQNS is limited to exponentially distributed service times and inter-arrival times. Furthermore, the support for analyzing blocking behavior is limited (Brosig et al., 2014). While approximations of service times are considered acceptable for mean-value analysis (e.g., Menascé et al. (1994)), approximations of the inter-arrival time distribution in case of an open workload may easily lead to considerable prediction errors (see, e.g., Section 7.2). In other words, LQNS's blocking behavior limitation and its limitation to exponentially distributed inter-arrival times are considered to introduce more significant inaccuracies than the limitation to exponentially distributed service times. Thus, the criteria to decide between (ii) and (iii) are then given by:

- If for all requested response times only their average is requested, and the performance model does not contain any acquire and release actions to model software contention, and *UsgSc1* has a closed workload, and mapping function $w_i \mapsto ((i-1) \text{ div } \lceil K/3 \rceil)$ maps $dw = w_i$ to a value greater or equal to 1, then choose LQNS.
- If for all requested response times only their average is requested, and the performance model does not contain any acquire and release actions to model software contention, and *UsgSc1* has an open workload with an exponentially distributed inter-arrival time, and mapping function $w_i \mapsto ((i-1) \text{ div } \lceil K/3 \rceil)$ maps $dw = w_i$ to a value greater or equal to 1, then choose LQNS.
- If for all requested response times only their average is requested, and mapping function $w_i \mapsto ((i-1) \text{ div } \lceil K/3 \rceil)$ maps $dw = w_i$ to a value equal to 2, then choose LQNS.
- In all other cases proceed with model solving technique (iii).

While, e.g., the bounds analysis does not have additional degrees-of-freedom, the transformation to QPN and solving with SimQPN provides multiple configuration options. These configuration options can be used to further tailor the performance prediction to the given query *DM* and *dw*.

- SimQPN provides fine-grained options to control what type and amount of data is logged during the simulation run. The more data is logged, the longer the simulation run takes. The logging configuration can be used to tailor the simulation to the demanded metrics *DM*. For each place, SimQPN allows the specification of a so-called *stats-level*. The stats-level indicates what statistics are collected at that place. Five levels are distinguished:
  - Level "no statistics". No statistics are collected at the place.
  - Level "throughput". Per token color, throughput statistics are collected at the place.
  - Level "utilization". If the place is a queueing place, the queue's average utilization is obtained.
  - Level "mean residence time". Per token color, mean token residence times are collected at the place.
  - Level "residence time distribution". Per token color, the empirical distribution of the residence time is collected at the place.
- SimQPN allows configuring various simulation stopping criteria. These criteria determine, e.g., at which confidence level the simulation stops. Thus, the stopping criteria provide an important degree-of-freedom that can be considered to tailor the solving technique to the given trade-off specification *dw*.

In the following, we describe how the above-mentioned degrees-of-freedom are used for tailored model solving. Figure 5.21 shows how SimQPN logging options are chosen depending on *DM*. For a $dm_i = (l_i, m_i, a_i) \in DM$, there is a decision tree showing which stats-level to select. The first decision depends on demanded metric m_i. The tree supports the metric types response time, throughput and utilization.

- If a different metric type is requested, the stats level of the QPN place corresponding to l_i is set to level "no statistics".
- If utilization is requested, l_i has to refer to a resource. The QPN places representing this resource are then annotated with level "utilization".
- If throughput is requested, the places representing l_i – which can be a resource or a service – are set to level "throughput".
- If a service response time is requested, the stats-level to choose depends on the aggregation a_i. If the average response time of a service is requested, a dedicated measurement place is introduced. Figure 5.22 illustrates the integration of a measurement place in the QPN resulting from a transformation of a call to l_i (see also Figure 5.12). Transition *Call-Entry* does not only fire tokens of color c to *Behavior-Begin-Place* but also to the newly introduced *Measurement-Place*. Transition *Call-Exit* may only fire a token to the *Call-End-Place* if both the *Behavior-End-Place* and the *Measurement-Place* contain a token of color c. In addition to the formal description of the call transformation in Section 5.3.1, the formal description of the added *Measurement-Place* is:

$$\begin{aligned}
&P := P \cup \{p_{\textit{Measurement-Place}}\}\\
&F_C(p_{\textit{Measurement-Place}}) := \{c\}\\
&(I^+(p_{\textit{Measurement-Place}}, t_{\textit{Call-Entry}}))(m_{t_{\textit{Call-Entry}},1}) := \{c\}_{MS}\\
&(I^-(p_{\textit{Measurement-Place}}, t_{\textit{Call-Exit}}))(m_{t_{\textit{Call-Exit}},1}) := \{c\}_{MS}\ .
\end{aligned}$$

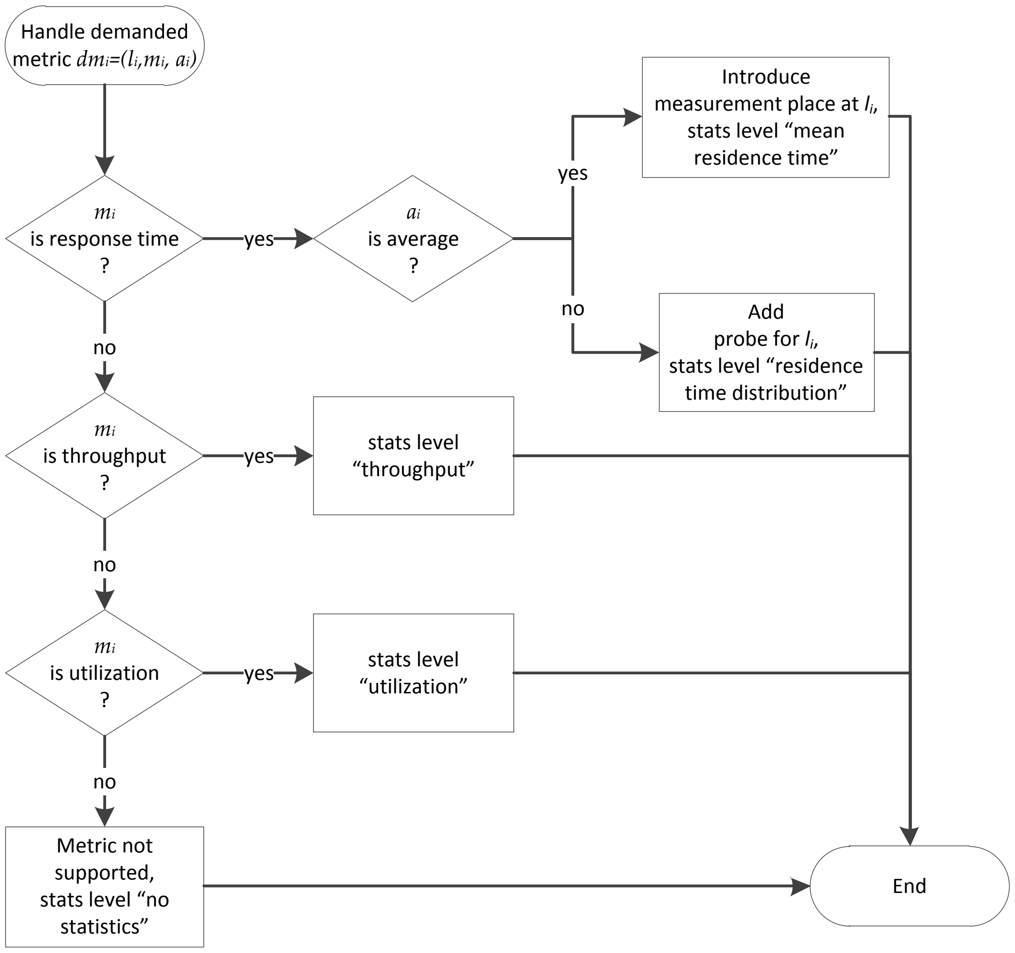

Handle demanded metric $dm_i=(l_i,m_i, a_i)$
m_i is response time ?
yes
a_i is average ?
yes
Introduce measurement place at l_i, stats level “mean residence time”
no
Add probe for l_i, stats level “residence time distribution”
no
m_i is throughput ?
yes
stats level “throughput”
no
m_i is utilization ?
yes
stats level “utilization”
no
Metric not supported, stats level “no statistics”
End

Figure 5.21: Decision Tree for Tailoring the SimQPN Configuration

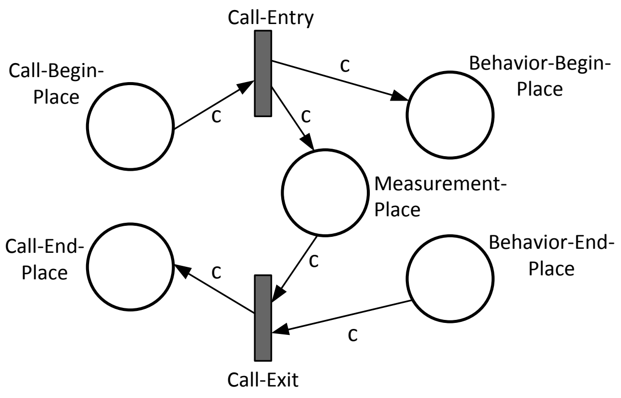


Figure 5.22: Measurement Place for a Call in a QPN

The mean residence time for tokens of color c at the *Measurement-Place* then corresponds to the average response time of service l_i.

If a percentile of the response time or the empirical distribution characterized as sample set are requested, the simulation needs to track requests, respectively tokens, as they are propagated through the QPN. For this purpose, SimQPN implements a so-called *probe* feature. A *probe* allows the tracking of tokens of a specified token color between a start and an end place. A token receives two additional properties, a timestamp and a probe identifier. The timestamp indicates the time when the token entered the start place. At the end place, the difference between the token's timestamp and the timestamp when the token entered the end place can be logged. In addition to the grouping per token color code the probe feature thus allows grouping tokens per probe id. However, note that individual tokens still do not have an identity. Using probes and annotating them with stats-level "residence time distribution", the service response time distribution can be approximated with the distribution of the measured probes.

Furthermore, SimQPN's simulation stopping criteria are configured to reflect trade-off specification dw. SimQPN uses non-overlapping batch means (Schmeiser, 1982) for estimating the variance of mean residence times and stops when a certain confidence level is reached. The significance level and the desired width of the confidence interval are configurable. The significance level is set to $\alpha = 0.05\%$, the width of the confidence interval is set relative to the mean value, i.e., as relative precision of the estimate. The relative precision is varied depending on the trade-off specification dw. The higher the precision, the higher the prediction accuracy, but also the longer the prediction run. For $dw = w_\top$ (trade-off weight with highest accuracy), the relative precision is set to 5%. For $dw = w_\perp$ (trade-off weight with fastest prediction speed), the relative precision is set to 30%. In capacity planning, prediction errors of up to 30% are considered acceptable (Menasce and Virgilio, 2000). Accordingly, a trade-off specification $dw = w_i \in W$ then maps to a relative precision of $(K - i) * (30 - 5)/K + 5$ percent.

The output of the model solving step are the predictions for the demanded metrics. The prediction itself is tailored to the given performance query in order to provide the demanded metrics in accordance with the given specification of how to trade-off between prediction accuracy and time-to-result. Accurate predictions are obtained using detailed simulation. To optimize the prediction speed, the simulation stopping criteria as well as type and amount of simulation log data are configured appropriately. Faster predictions, however, are obtained using established analytical solvers such as LQNS, or analytical techniques such as approximative bounds analysis.

5.5 Performance Queries

In this section, we generalize the notion of a performance query and provide a declarative interface to performance prediction techniques to simplify and automate the process of using architecture-level software performance models for performance analysis. The proposed language named DQL provides means to express the demanded performance metrics for prediction as well as the goals and constraints in a specific prediction scenario (Gorsler et al., 2014). It is not limited to online scenarios; it constitutes a useful tool also for querying design-time performance models.

5.5.1 Requirements

We investigate common usage scenarios in performance engineering (see introduction to Chapter 5) to derive user stories as requirements for our language for expressing performance queries. It is distinguished whether performance queries are issued in an *online* or *offline* context, i.e., if the system is at run-time during operation or if the system is in the development or deployment phase. The following user stories (Cohn, 2004) are formulated as requirements for the query language.

As a user,

- I want to issue queries independent of the underlying performance modeling formalism.
- I want to list the modeled services of a selected performance model instance.
- I want to list the modeled resources of a selected performance model instance.
- I want to list the variable parameters of a selected performance model instance.
- I want to list supported performance metrics for selected services and resources.
- I want to conduct a prediction of selected performance metrics of selected services and resources.
- I want to aggregate retrieved performance metrics by statistical means.
- I want to control the performance prediction by specifying a trade-off between prediction speed and accuracy.
- I want to conduct a sensitivity analysis for selected parameters in defined parameter spaces.
- I want to query revisions of model instances.

We emphasize that the query language needs to be independent of a specific performance modeling formalism. Predictable performance metrics may vary from model instance to model instance and model solver to model solver, so the query mechanism needs to include means to evaluate the underlying model and list supported performance metrics to the user. For each performance modeling formalism, the query language requires a *Connector* to bridge the mentioned gaps. Furthermore, queries should be user-friendly to write, e.g., there should be a text editor with syntax highlighting and auto-completion features.

5.5.2 Performance Query Language

In this section, we present the concepts of the performance query language. We use syntax diagrams to describe the most relevant parts of the language grammar. Details can be found in the master's thesis of Gorsler (2013) that has been supervised by the author. Figure 5.23 shows the uppermost grammar rule as a syntax diagram also referred to as railroad diagram. In general, there are two query classes: (i) A ModelStructureQuery is used to analyze the structure of performance models. It can provide information about available services and resources, performance metrics as well as model variation points. (ii) A PerformanceMetricsQuery is used to trigger actual performance predictions. Both query classes are followed by a ModelAccess part that refers to a performance model instance.

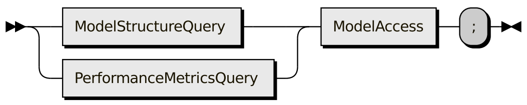


Figure 5.23: Query Classes of Descartes Query Language (DQL)

5.5.2.1 Model Access

The query language is independent of a specific performance modeling formalism. Thus, to issue a query on a performance model instance, both the location of the model instance as well as a *DQL Connector* needs to be specified. A DQL Connector is specific for a performance modeling formalism and bridges the gap between the performance model and DQL. Figure 5.24 shows the model access initiated by the keyword USING. The nonterminal

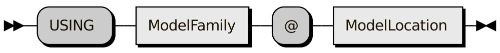


Figure 5.24: Model Access

ModelFamily refers to an identifier that serves as a reference to a DQL Connector. The DQL Connector has to be registered in a central DQL Connector registry (see Section 5.5.3). The ModelLocation is a reference to a model instance location.

5.5.2.2 Model Structure Query

The user can request information about which services or resources are modeled, and for which model entities the referenced model instance can provide which performance metrics. In DQL notation, a model entity is either a resource or a service. Figure 5.25

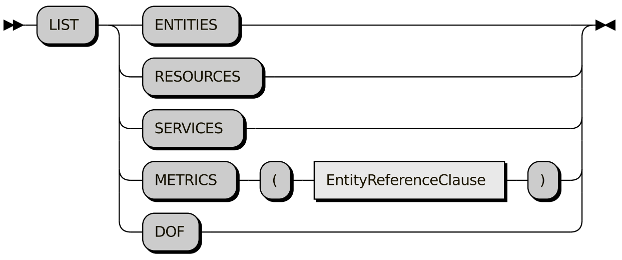


Figure 5.25: Model Structure Query

shows a ModelStructureQuery initiated by the keyword LIST. Using terminals ENTITIES, RESOURCES, SERVICES the user can query for respective entities, resources or services. The result is a list of entity identifiers that are unique for the referenced model instance. Listing 5.1 illustrates a simple query example.

```
LIST ENTITIES
USING connector@location;
```

Listing 5.1: List all Modeled Entities

Besides querying for services and resources, the user can also query for supported performance metrics. We denote a performance metric as *available* if the performance metric can be derived from the performance model instance. For example, for a CPU resource of an application server, the average utilization is typically available. Since the available performance metrics may differ from entity to entity, the user has to specify for which entity the available performance metrics should be listed. For that purpose, the user has to provide an EntityReferenceClause. An EntityReferenceClause is a comma-separated list of EntityReferences whose syntax is illustrated in Figure 5.26.

An entity reference thus starts with a keyword identifying the entity type (RESOURCE or SERVICE) followed by an entity identifier and an optional AliasClause. Listing 5.2 shows a corresponding query example. In the example, the user queries for available metrics of a

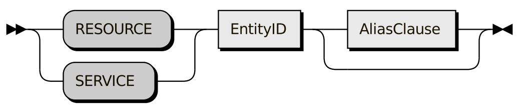


Figure 5.26: Entity Reference

resource with identifier 'AppServerCPU1' and a service with identifier 'newOrder'. Alias `r` is assigned to the resource and alias `s` is assigned to the service.

```
LIST METRICS
     (RESOURCE 'AppServerCPU1' AS r, SERVICE 'newOrder' AS s)
USING connector@location;
```

Listing 5.2: List available Performance Metrics

Furthermore, DQL allows querying for model variation points, also denoted as Degrees-of-Freedom (DoFs). The query result then is a list of DoF identifiers. The way how model variation points are modeled is independent from DQL, it depends on the DQL Connector. We provide an example of DoF queries in Section 5.5.2.3.

5.5.2.3 Performance Metrics Query

A PerformanceMetricsQuery is used to trigger performance predictions. Figure 5.27 shows the syntax. First, we explain the parts of the query that are obligatory to write basic queries. For optional extensions such as query constraints, evaluations of DoFs and model revisions, we refer to subsequent paragraphs.

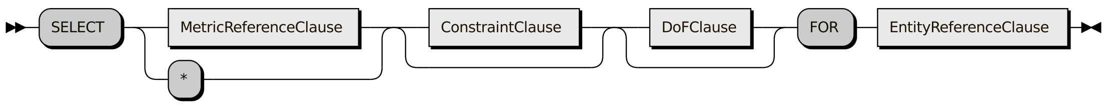


Figure 5.27: Performance Metrics Query

A user can specify the performance metrics of interest using a wildcard '*' (all available performance metrics) or via the nonterminal MetricReferenceClause. MetricReferenceClause is a comma-separated list of MetricReferences shown in Figure 5.28. A MetricReference either refers to a single metric or to an aggregated metric. A single metric is described by an EntityIdOrAlias followed by a dot and a MetricId or wildcard. Listing 5.3 shows a corresponding example where the utilization of an application server CPU and the average response time of a service is requested.

A specification of an aggregated metric consists of two parts. The first part (nonterminal AggregateFunction) selects an aggregate function. The set of supported aggregate functions is based on the descriptive statistics part of Apache Commons Math[4] and provides common statistical means, e.g., arithmetic and geometric mean, percentiles, sum and variance. The second part describes the list of performance metrics that should be aggregated. A wildcard ('*') can be used to iterate over all entities where a specific performance metric is available. An exemplary use of an aggregated metric is shown in Listing 5.4, where the mean value of two application server utilization rates is computed. Note that the computation of the aggregate is provided by the DQL Query Execution Engine (QEE) (see Section 5.5.3) and

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

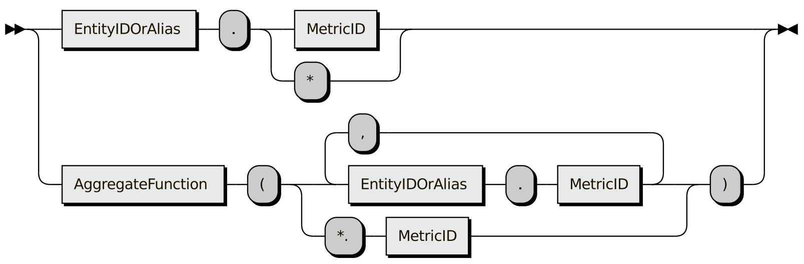


Figure 5.28: Metric Reference

```
SELECT r.utilization, s.avgResponseTime
FOR RESOURCE ‘AppServerCPU1’ AS r, SERVICE ‘newOrder’ AS s
USING connector@location;
```

Listing 5.3: Trigger Basic Performance Prediction

is *not* part of the DQL Connector. The DQL Connector only needs to support querying individual metrics.

Constraints

As motivated in Section 5.4, in online performance and resource management scenarios, controlling performance predictions by specifying a trade-off between prediction accuracy and time-to-result is important to act in time (Thereska et al., 2005; Kounev et al., 2010). DQL enables the specification of such constrained performance queries. The syntax of the corresponding ConstraintClause is shown in Figure 5.29. The ConstraintIDs are DQL Connector specific and are intended to control the behavior of the underlying model solving process. For instance, a DQL Connector might support a constraint named `‘FastResponse’` to trigger fast analytical mean-value solvers, or a constraint named `‘Detailed’` to trigger a full-blown simulation that may take a significant amount of time simulating, e.g., fine-grained OS-specific scheduling behavior (Happe et al., 2010). Listing 5.5 shows a corresponding example.

Degrees-of-Freedom

If the user wants to optimize service compositions or configuration parameter settings, an automated design and parameter space exploration covering defined DoFs is helpful (Koziolek and Reussner, 2011; Koziolek et al., 2013; Huber et al., 2012a). DoFs specify how entities in a performance model can be varied and thus span the space of valid configurations and parameter settings. Depending on the size of the configuration space and the space exploration strategy, the time-to-result of a single performance prediction gains in importance. Otherwise, the exploration might take too long to be feasible.

To evaluate DoFs, DQL provides several optional language constructs. Figure 5.30 shows the syntax diagram of nonterminal DoFClause. A DoFClause refers to DoFs (nonterminal DoFReferenceClause) and an optional exploration strategy.

A DoFReferenceClause is a comma-separated list of nonterminal DoFReference that is shown in Figure 5.31. It starts with a DoF identifier (with an optional alias) and is followed by DoFVariationClause that provides optional parameter settings (see Figure 5.32). In its current version, DQL supports lists of parameter values of type `Integer` or `Double` as well as interval definitions of type `Integer`. Listing 5.6 shows an example with two DoFs. On the one hand, we vary the inter-arrival time of the open workload (values

```
SELECT MEAN(r1.utilization, r2.utilization)
FOR RESOURCE 'AppServer1' AS r1, RESOURCE 'AppServer2' AS r2
USING connector@location;
```

Listing 5.4: Query with Aggregated Metric

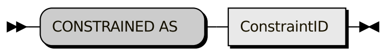


Figure 5.29: Constraint Clause

0.1, 0.2, 0.3), on the other hand, we vary the size of the database connection pool from 10 to 30 in steps of 5. Without an explicitly defined exploration strategy, the default exploration strategy is considered to be a full exploration. In the example, this means that $3 \times 5 = 15$ performance predictions are triggered. The query result set is then a list of 15 prediction results. Each prediction result contains the prediction for performance metrics `r.utilization` and `s.avgResponseTime`.

Using the optional ExplorationStrategyID, together with user-defined configuration properties (see Figure 5.33), it is possible to trigger an alternative exploration strategy provided that the DQL Connector supports it. This is necessary for, e.g., DoFs representing migrations of Virtual Machines (VMs) from a physical host machine to another physical host machine. In these cases, it is the DQL Connector that needs to provide means to iterate the configuration space. An integration of complex exploration strategies, e.g., multi-attribute Quality of Service (QoS) optimization techniques to derive Pareto-optimal solutions (Koziolek et al., 2013), is thus supported. Listing 5.7 shows a query example with an explicit exploration strategy. The query has one DoF, namely the physical machine where the application server VM is deployed. The configuration space, a set of two physical machines, is described as String value of property `targets`. Note that the semantics of the configuration properties are specific for the DoF and the exploration strategy, i.e., the properties are not interpreted by DQL.

Temporal Dimension

As additional feature for Performance Metrics Queries, DQL offers facilities to access different revisions of a performance model. The assumption is that the model instances are annotated with a revision number and/or a timestamp, i.e., that there is a chronological order.

In particular, if the performance models are used in online scenarios, queries that allow the user to ask about performance metrics in the past, the development of performance metrics over time, or (together with a workload forecasting mechanism (Herbst et al., 2013)) the anticipated development of performance metrics, are desirable. In online scenarios, performance model instances are typically part of a *performance data repository* that integrates revisions of calibrated model instances and performance monitoring data (Kounev et al., 2010).

DQL allows to express the temporal dimension in different ways: (i) with a time frame defined by a start and end time, and (ii) with a time frame starting or ending with the current time and a time delta. Alternative (i) is used in the example in Listing 5.8. Resource utilization and service response time are queried for a specific time frame of one day. The results are sampled groups of one hour, possibly read from historical monitoring data, thus leading to a set of 24 result sets. The example shown in Listing 5.9 uses alternative (ii).

```
SELECT r.utilization, s.avgResponseTime
CONSTRAINED AS ‘FastResponse’
FOR RESOURCE ‘AppServerCPU1’ AS r, SERVICE ‘newOrder’ AS s
USING connector@location;
```

Listing 5.5: Constrained Query

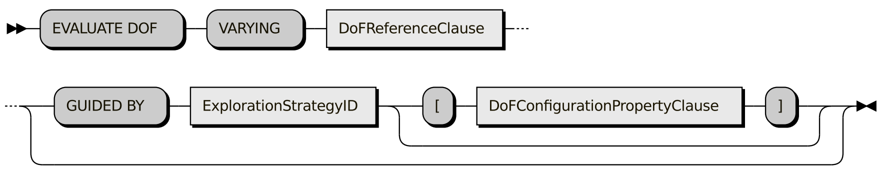


Figure 5.30: DoF Clause

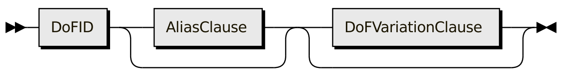


Figure 5.31: DoF Reference

```
SELECT r.utilization, s.avgResponseTime
EVALUATE DOF
  VARYING
    ‘DoF_OpenWorkload_InterarrivalTime’ AS dof1 <0.1, 0.2, 0.3>,
    ‘DoF_JDBCConnectionPool_Size’ AS dof2 <10..30 BY 5>
FOR RESOURCE ‘AppServerCPU1’ AS r, SERVICE ‘newOrder’ AS s
USING connector@location;
```

Listing 5.6: DoF Query

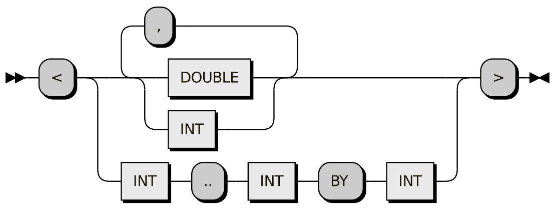


Figure 5.32: DoF Variation Clause

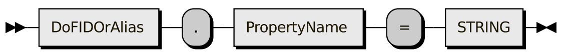


Figure 5.33: DoF Configuration Property

```
SELECT s.avgResponseTime
EVALUATE DOF
  VARYING ‘DoF_AppServerVM_Migration’ AS dof1
  GUIDED BY ‘MyExplorationStrategy’
    [dof1.targets = ‘‘PhysicalMachineA,PhysicalMachineB’’]
FOR SERVICE ‘newOrder’ AS s
USING connector@location;
```

Listing 5.7: DoF Query with Exploration Strategy

The query requests the application server CPU utilization for the next two hours, sampled in twelve groups of ten minutes length each. This query triggers performance predictions, provided that a workload forecast for the next two hours is available.

```
SELECT r.utilization, s.avgResponseTime
FOR RESOURCE 'AppServerCPU1' AS r, SERVICE 'newOrder' AS s
USING connector@location
OBSERVE
  BETWEEN '2013-10-09 08:00:00' AND '2013-10-10 08:00:00'
  SAMPLED BY 1h;
```

Listing 5.8: Performance Metrics Over Time

```
SELECT r.utilization
FOR RESOURCE 'AppServerCPU1' AS r
USING connector@location
OBSERVE
  NEXT 2h SAMPLED BY 10M;
```

Listing 5.9: Anticipated Resource Utilization

5.5.3 Architecture

Based on the query language as described in the previous sections, we propose an implementation of DQL based on an extensible software architecture. The component-based

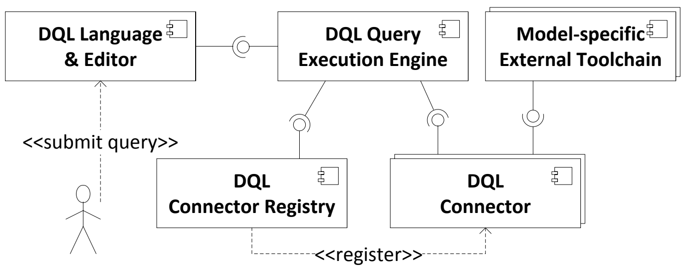


Figure 5.34: DQL System Architecture, cf. Gorsler et al. (2014)

architecture of the DQL approach is shown in Figure 5.34.

The first component, *DQL Language & Editor*, provides the interface to users and offers an Application Programming Interface (API). The component provides a DQL query parser and represents statements in a model of the abstract syntax tree. For convenience, a text editor is also part of DQL. It provides code assistance to obtain identifiers of model entities or available performance metrics. Furthermore, users can issue queries and visualize query results. The second component, *DQL Query Execution Engine (QEE)*, provides the main execution logic in the DQL system architecture. Here, all tasks that are independent of specific performance modeling formalisms and prediction techniques are implemented. The DQL QEE selects an adequate DQL Connector to access the requested model instance, to execute the query, and to provide the results to the user. The DQL QEE also calculates aggregation functions if requested and performs the necessary pre- and post-processing steps. The third component, a *DQL Connector*, provides functionality that is specific to the employed performance modeling formalism and prediction technique. This includes accessing performance models, triggering the prediction process and providing additional information, e.g., about the model structure and the available performance metrics. To integrate different approaches for performance prediction in a DQL environment, multiple DQL Connectors can be deployed. As each DQL Connector comes with a unique identifier that needs to be referenced in a DQL query together with a model location, the DQL QEE can select a suitable DQL Connector to execute the query. To find a suitable DQL Connector, the *DQL Connector Registry* is used which maintains an index of available DQL Connectors and their support for query classes.

The proposed architecture allows to integrate various different performance modeling approaches and prediction techniques. Extensions of the DQL environment are primarily possible by contributing additional DQL Connectors. Note that DQL Connector implementations, due to the modular language structure, do not need to implement the entire feature set of DQL, i.e., it is allowed to implement approaches that are not capable of all DQL features.

There is a *DML Connector* providing performance prediction functionality for the performance modeling abstractions described in Chapter 4. The performance metrics the connector supports are depicted in Section 5.4.1, the allowed constraints correspond to the trade-off specifications introduced in Section 5.4.2.

5.6 Summary

This chapter showed how to conduct online performance predictions using the performance modeling abstractions that are described in Chapter 4. Figure 5.35 provides an overview of the prediction process showing the individual steps and their inputs and outputs. We assume an architecture-level performance model to be available. The actual prediction process is triggered by a performance query. The query indicates which performance metrics are to be predicted and includes a specification of how to trade-off between prediction accuracy and prediction overhead.

The model composition step marks those parts of the architecture-level performance model that need to be considered to answer the query. These markings are kept in a composition mark model which serves as input for the pre-processing step where parameter dependencies are solved and model variables are characterized. This step, described in Section 5.2, (i) resolves the probabilistic parameter dependencies introduced in Section 4.1.5 and (ii) parameterizes the performance model on-the-fly using the monitoring interface introduced in Section 4.1.6. The output is a callstack model that keeps the call graph together with the corresponding model parameter values. The call graph determines how the performance model has to be traversed for the performance prediction.

The subsequent step is the tailored model solving step that solves the performance model, i.e., predicts the requested metrics under consideration of the given trade-off specification. We use existing model solving techniques based on established modeling formalisms, namely bounds analysis (Menasce and Virgilio, 2000), LQNs (Franks et al., 2009), and QPNs (Kounev and Buchmann, 2006) (Section 5.3). The challenge is to find a balance between prediction accuracy and prediction overhead (Section 5.4). The results of the tailored model solving step are then returned to the query issuer.

The chapter concluded with Section 5.5 where the notion of a performance query is formalized. It provided a generic declarative interface to performance prediction techniques to simplify and automate the process of using architecture-level software performance models for performance analysis. The proposed DQL is a language to express the demanded performance metrics for prediction as well as the goals and constraints in a specific prediction scenario.

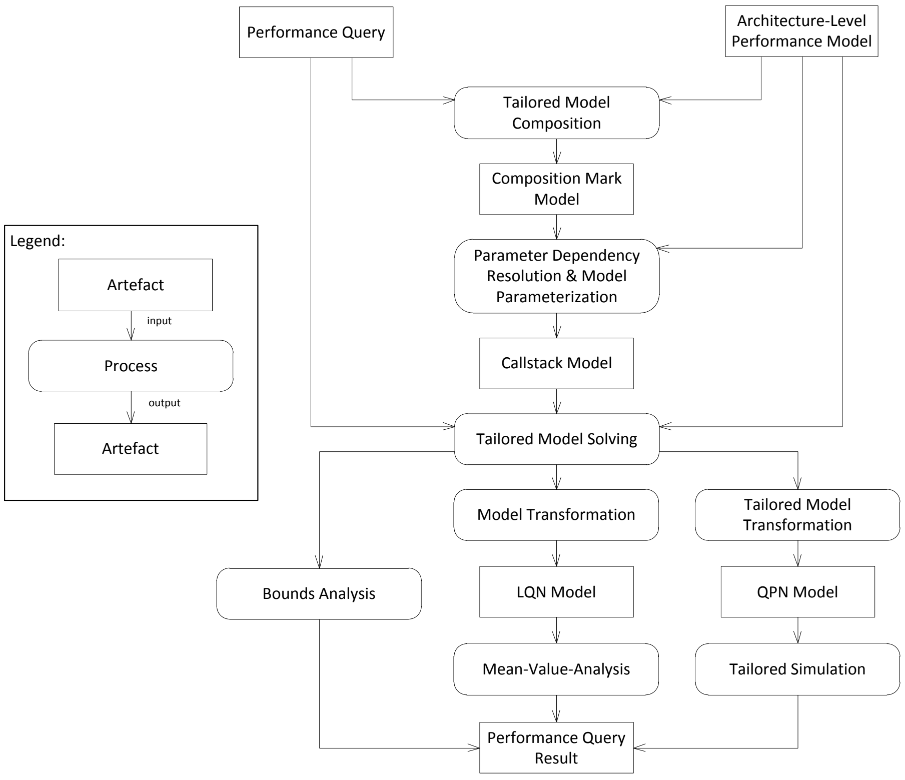


Figure 5.35: Online Prediction Process

6. Integration of Architecture-Level Performance Models and System Environments

For online performance predictions, it is essential to keep the performance model in sync with the modeled system. Otherwise, once a performance model of the system is built, the performance model can quickly become outdated and would thus not be representative of the real system anymore (Brosig, 2011). Configuration and deployment changes are common in modern enterprise system environments. For instance, new services are deployed on-the-fly, service compositions are changed, virtual machines are migrated or servers are consolidated during operation (Kounev et al., 2010). If the system changes, the performance model representations must also be updated.

Our approach is to *tie* the performance model to the system environment, i.e., to continuously adapt the model during system operation. The model should provide up-to-date and exact information about the system to enable accurate performance predictions. In the terms of the models@runtime community (Blair et al., 2009), the model should be a "causally connected self-representation of the associated system" (Blair et al., 2009) such that it constantly *mirrors* the performance-relevant structure and behavior of the system.

In order to achieve such a *mirroring*, we follow the proposal of Woodside et al. (2007) of a convergence of performance monitoring, modeling and prediction as interrelated activities. At run-time, the system components are implemented and deployed in the target production environment. This makes it possible to obtain representative estimates of the various model parameters taking into account the real execution environment. Moreover, model parameters can be continuously adjusted to iteratively refine their accuracy. Furthermore, performance-relevant information can be monitored and described at the component instance level and not only at the type level as typical for performance models at design-time. However, during operation, we do not have the possibility to run arbitrary experiments since the system is in production and is used by real customers issuing requests. In such a setting, monitoring has to be handled with care, keeping the monitoring overhead within limits such that system operation is not disturbed.

In this chapter, we develop methods to integrate architecture-level performance models and system environments. The integration is realized by: (i) a technique to extract model instances semi-automatically based on monitoring data, and (ii) a technique to automatically maintain the extracted instances at run-time. In each case, we distinguish between static structural information of the system environment (e.g., involved component types) and dynamic parameters (e.g., resource demands) that are reflected in the models. Figure 6.1 illustrates the concept. Both semi-automatic extraction as well as model maintenance use monitoring data collected at run-time as input to extract/maintain an architecture-level performance model.

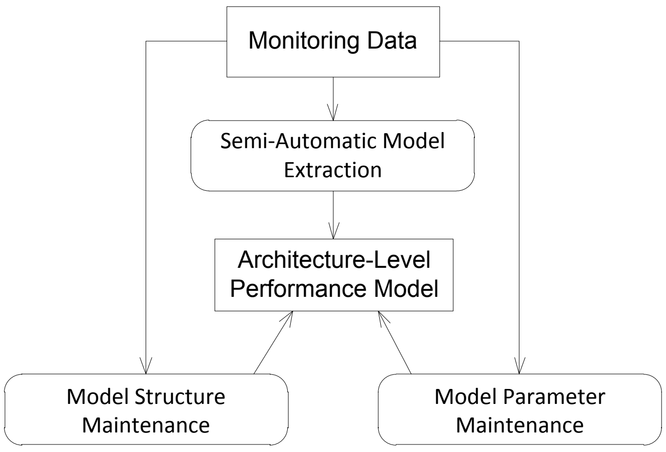

Monitoring Data
Semi-Automatic Model Extraction
Architecture-Level Performance Model
Model Structure Maintenance
Model Parameter Maintenance

Figure 6.1: Extraction and Maintenance of Architecture-Level Performance Models

The chapter is structured as follows: Section 6.1 describes required monitoring capabilities of the monitoring infrastructure. Section 6.2 describes the semi-automatic extraction of architecture-level performance models based on monitoring data. Section 6.3 describes how the model structure is maintained. In Section 6.4, the important question of how to obtain model parameter values is answered. Section 6.4 describes how model parameters such as resource demands can be maintained and how parameter dependencies can be probabilistically characterized. Section 6.5 discusses how architecture-level performance model parameters can be calibrated and adjusted in order to increase their accuracy. Section 6.6 gives a summary of this chapter.

6.1 Monitoring Capabilities

In this section, we describe which specific capabilities the underlying monitoring infrastructure needs to support in order to be able to integrate architecture-level performance models and system environments.

Besides common monitoring features such as monitoring resource utilization, the monitoring infrastructure should allow the tracking of system requests (see Section 6.1.1) and provide facilities to control the overhead incurred by the monitoring operations (see Section 6.1.2).

Furthermore, given that measurement data is used to characterize model parameters such as response time, control flow or resource demand descriptions, it should be possible to aggregate measured empirical distributions in a compact fashion. For instance, it is not feasible to characterize a loop iteration count distribution with thousands of measurement samples. Instead, the distribution should be abstracted in a Probability Mass Function (PMF) that is manageable in a performance prediction process as described in Section 5. Techniques to aggregate measurement data are shown in Section 6.1.3.

6.1.1 Call Path Tracing

System requests can be tracked using a technique we denote as *call path tracing*, as described in the following. An executed system request translates into a path through a control flow graph whose edges are basic blocks (Allen, 1970), i.e., an edge represents a portion of code within an application with only one entry point and only one exit point. A path through the control flow graph can be represented by a sequence of references to basic blocks. For the sake of simplicity, requests with forking behavior are neglected, such requests would translate to a tree in the control flow graph.

We assume we can instrument the system to monitor so-called *event records.*

Definition. *An event record is defined as tuple $e = (l, t, s)$ where l refers to the begin or end of a basic block, t is a timestamp and s identifies a system request. The event record indicates that l has been reached by s at time t.*

In order to *trace* individual system requests, a set of event records has to be obtained at run-time. The set of gathered event records then has to be (i) partitioned and (ii) ordered. The set of event records is partitioned in equivalence classes $[a]_\mathcal{R}$ according to the following equivalence relation:

Definition. *$\mathcal{R}$ is a relation on event records: Let $a = (l_1, t_1, s_1), b = (l_2, t_2, s_2)$ be event records obtained through instrumentation. Then a relates to b, i.e., $a \sim_\mathcal{R} b$, if and only if $s_1 = s_2$.*

Ordering the event records of an equivalence class in chronological order then leads to a sequence of event records, and thus can be used to derive a call path trace. We refer to Briand et al. (2006); Hrischuk et al. (1999); Israr et al. (2007); Anderson et al. (2009) where call path traces are transformed to, e.g., UML sequence diagrams.

6.1.2 Overhead Control

We use monitoring data to semi-automatically extract architecture-level performance models and to keep them up-to-date during system operation. However, the monitoring overhead has to be kept low for two reasons: First, given that the system is observed during operation, measurements must not cause significant performance degradations. Second, the monitoring overhead may disturb model parameter maintenance in a way that, e.g., estimated resource demands are inaccurate and biased (see Section 6.4.3).

To reduce the overhead of monitoring system requests, in general there exist two orthogonal concepts: (i) *quantitative throttling*: throttling the number of requests that are actually monitored, (ii) *qualitative throttling*: throttling the level of detail the requests are monitored at. Existing work on (i) is presented, e.g., in Gilly et al. (2009). The authors propose an adaptive time slot scheduling for the monitoring process. The monitoring frequency depends on the load of the system. In phases of high load, the monitoring frequency is throttled. Concerning (ii), the monitoring approach presented in Ehlers and Hasselbring (2011) allows an adaptive monitoring of requests, i.e., monitoring probes can be enabled or disabled depending on what information about the requests should be monitored.

6.1.3 Empirical Characterizations

Sets of measurement samples are used to characterize model parameters such as response time, control flow or resource demand descriptions. The descriptions are modeled as RandomVariables, each of them represented by a probability distribution over the sample space. Given that the sample spaces of response times, loop iteration counts, and resource demands are infinite, and in case of response times and resource demands also continuous, a PMF attributing each observation an individual probability is infeasible if a large number of samples is on hand. Instead of using a PMF to characterize the random variable, one has to approximate a probability density function. An exemplary approximation f_X' for a density function $f_X(x)$ of a random variable X is, e.g.,

$$f_X'(x) = \begin{cases} 0.0 & x < 10.0, \\ 0.04 & 10 \leq x < 30.0, \\ 0.1 & 30 \leq x < 32, \\ 0.0 & 32 \leq x. \end{cases}$$

As Stochastic Expression (StoEx) this is denoted as `DoublePDF[(10.0;0.0)(30.0;0.04)(32.0;0.1)]` (see Section 4.1.4.3).

In other words, given a set of measurement samples, a histogram representing the empirical density function needs to be build. The number of histogram bins and their sizes need to be chosen in order to simplify the representation of the distribution on the one hand, while still providing a representative shape of the density function on the other hand. Many approaches to build histograms are available. One common application area are databases: "Database systems maintain histograms to summarize the contents of relations and permit efficient estimation of query result sizes and access plan costs" (Poosala et al., 1996). There are approaches that base the bin sizes on the range of the measurement samples (Sturges, 1929), build the histogram under the assumption of a normal distribution (Scott, 1979), or are distribution-independent and based on the range of quartiles (Freedman and Diaconis, 1981). Furthermore, we distinguish between *static* histograms, i.e., the underlying sample set has to be available at the beginning of a histogram's construction, and *dynamic* histograms, i.e., a histogram can be iteratively refined as new measurement samples become available (Gibbons et al., 2002). For an overview of histogram types and a histogram taxonomy we refer to Poosala et al. (1996).

6.2 Semi-Automatic Model Extraction

The goal is to automate the process of building performance models by observing the system behavior at run-time. Performance-relevant abstractions and parameters should be automatically extracted using monitoring data with as little human intervention as possible. This section is based on the work presented in Brosig et al. (2011) and focuses on extracting the architecture-level performance abstractions introduced in Chapter 4. More specifically, we provide a technique to extract the application architecture described in Section 4.1. Information about how to extract information about the resource landscape (see Section 4.2) can be found in Huber et al. (2011b, 2010). Usage profile extraction from log data is covered in Hoorn et al. (2008); van Hoorn (2014a).

The process we employ to extract performance-relevant abstractions of the application architecture includes four main steps depicted in Figure 6.2. First, the *effective* application architecture (Israr et al., 2007) is extracted, i.e., the set of components and connections between components that are effectively used at run-time. Second, the service behavior abstractions are extracted. Third, performance model parameters are extracted. Fourth, the resulting model is iteratively adjusted until it provides an *acceptable accuracy*. The set of methods to *extract* the model parameters is equivalent to the methods that are used for parameter *maintenance* and are thus presented in Section 6.4. Model calibration and adjustment as well as the notion of *acceptable accuracy* are discussed in Section 6.5.

In the following, we describe how the *effective* application architecture is extracted. Given a component-based application, extracting its architecture requires identifying its building blocks and the connections between them. Given that we use run-time monitoring data for the extraction, we consider only those parts of the architecture that are effectively

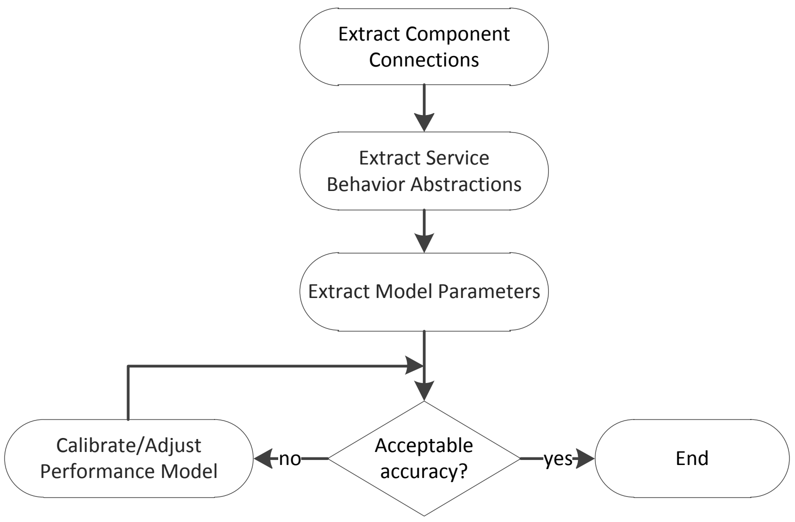

Extract Component Connections
Extract Service Behavior Abstractions
Extract Model Parameters
Calibrate/Adjust Performance Model
no
Acceptable accuracy?
yes
End

Figure 6.2: Model Extraction Process

used at run-time. Parts of the application architecture for which there is no run-time monitoring data available are thus neglected. This is in contrast to existing approaches for automated architecture-level performance model extraction such as Krogmann et al. (2010). Krogmann et al. (2010) extracts behavior models via dynamic *and* static analysis and relies on manual instrumentations.

6.2.1 Extraction of Component Connections

Componentization is the process of breaking down the considered application architecture into components. It is part of the mature research field of software architecture reconstruction (Tonella et al., 2007). Component boundaries can be obtained in different ways, e.g., specified manually by the system architect or extracted automatically through static code analysis (e.g., Chouambe et al. (2008)). The granularity of the identified components determines the granularity of the work units whose performance needs to be characterized, and hence the granularity of the resulting performance model. In the context of automated model extraction, a component boundary is specified as a set of software building blocks considered as a single entity. For instance, this can be a set of classes or a set of Java Servlets.

Once the component boundaries have been determined, the connections between the components can be automatically identified based on monitoring data. We determine the control flow between the identified components using *call path tracing* (see Section 6.1.1). Given a list of call paths and the knowledge about component boundaries, the list of effectively used components, as well as their actual entries (provided services) and exits (required services) can be determined (Brosig et al., 2011; Brosig, 2009; Brosig et al., 2009). Furthermore, for each component's provided service one can determine the list of external services that are called. Obviously only those paths that are exercised can be captured.

In terms of the application architecture meta-model presented in Section 4.1), we extract model entities Component, Interface, InterfaceProvidingRole and InterfaceRequiringRole. For a Component, its provided services constitute an Interface. This interface is then referred by the component via an InterfaceProvidingRole. Similarly, a component's required service induces an InterfaceRequiringRole, connecting the component to the Interface that contains the required service.

6.2.2 Extraction of Service Behavior Abstractions

After extracting the components and the connections between them, the service behavior abstractions introduced in Section 4.1.3 need to be extracted. We distinguish the three abstraction levels BlackBoxBehavior, CoarseGrainedBehavior and FineGrainedBehavior. The three levels require different types of monitoring data to be extracted.

Black Box Behavior

A BlackBoxBehavior captures the view of the service behavior from the perspective of a service consumer without any additional information about the service behavior. A BlackBoxBehavior model can be extracted by parameterizing it with a measured response time (see Section 6.4).

Coarse-Grained Behavior

A CoarseGrainedBehavior captures the component behavior when observed from the outside at the component boundaries. A CoarseGrainedBehavior can be extracted by parameterizing the frequency of external service calls and the overall service resource demands (see Section 6.4). Thus, we require information about the service's total resource consumption, however, no information about the service's internal control flow is assumed.

Fine-Grained Behavior

To describe a FineGrainedBehavior, we require information about the performance-relevant (component-internal) service control flow. The performance-relevant control flow consists of the service's internal resource demanding behavior, on the one hand, and information about how it makes use of external service calls, on the other hand. Obviously, it makes a difference if an external service is called once or, e.g., ten times within a loop. Furthermore, the ordering of external service calls and internal computations of the provided service may have an influence on the service performance. The FineGrainedBehavior model we aim to extract is an abstraction of the actual control flow. Performance-relevant actions are internal computational tasks and external service calls, hence also loops and branches where external services are called.

The set of call paths derived in the previous step (see Section 6.2.1) provides information on how a provided component service relates to external service calls of the component's required services. Formally, let X^C be the set of provided services of component C and $Y^C = y_1^C, \ldots, y_{m^C}^C$ the set of its required services. For compactness, we omit the index C from now on. Then, the observed call paths constitute a function $\mathcal{G} : x \mapsto S_x$ with $x \in X, S_x \subseteq S$ where $S = \{(l_1, \ldots, l_k) | k \in \mathbb{N}, l_i \in Y\}$. $\mathcal{G}$ maps x to S_x that represents the set of sequences of observed external service calls for provided service x. For instance, if there is a provided service x such that $\forall s_x \in \mathcal{G}(x) : s_x = (y_1) \vee s_x = (y_2)$ holds, then one could assume that service x has a control flow where either y_1 or y_2 is called, i.e., that there is a branch between two external calls to y_1 and y_2.

Multiple approaches exist for determining the performance-relevant control flow constructs, e.g., in Kappler et al. (2008); Krogmann et al. (2010). By instrumenting the performance-relevant control flow constructs inside the component explicitly, the control flow can be directly extracted from the obtained call paths (Brosig, 2009; Brosig et al., 2009, 2011).

Scopes of Model Variables

Having extracted the service behavior abstractions, the scopes of the identified ModelVariables can be set (see Section 4.1.4). In case a ModelVariable does not have a specified scope, which is the default case, the ModelVariable is globally unique. Monitoring data from all observations of the ModelVariable can then be used interchangeably among all instances of the component type that contains the ModelVariable.

The scope can be restricted by manually providing an appropriate ScopeSet, as described in Section 4.1.4.2. An automated extraction of a model variable's scope requires a top-down cluster analysis (Izenman, 2009) based on the observed values of the model variable. The goal is to find a clustering so that values observed at one component instance can be assigned to exactly one cluster. One cluster for all observed values corresponds to an empty ScopeSet of the model variable. One cluster for each component instance corresponds to a ScopeSet of the model variable that contains the variable's parent component type.

Influencing Parameters and Relationships

Furthermore, parameter dependencies can be added to the model. We propose to provide Relationships indicating parameter dependencies between InfluencingParameters and ModelVariables manually. Depending on whether a parameter dependency can be explicitly characterized, or should be just marked to be characterized empirically based on observations, its RelationshipCharacterizationType has to be configured (see Section 4.1.5). An automated detection of relevant parameter dependencies requires a sensitivity analysis of all involved parameters. Given that the set of all involved parameters contains all service input parameters and all ModelVariables, we consider an automated detection without a-priori knowledge to be infeasible. In particular, because during system operation we do not have the possibility to run arbitrary experiments (Kounev et al., 2010).

It is important to note that parameter dependencies are intended to improve the prediction accuracy when considering parameter variations, however, if they cannot be characterized or even if they are not explicitly modeled, a performance prediction can still be conducted.

6.3 Model Structure Maintenance

The performance model instance should be maintained during system operation, i.e., the instance should be kept up-to-date as the system evolves. If the system changes, the performance model representations must also be updated. In the terms of the models@runtime community (Blair et al., 2009), the model should be a "causally connected self-representation of the associated system" (Blair et al., 2009) such that it constantly *mirrors* the performance-relevant structure and behavior of the system.

In this section, we assume an architecture-level performance model instance to be available, and assume the instance to be representative for the modeled system in terms of its performance abstractions. Following the definition of Swanson (1976), we distinguish three types of possible system changes, namely corrective changes, adaptive changes and perfective changes. Briefly, corrective changes have the goal to correct software defects. Adaptive changes are performed to adapt the software system to changes in the environment such as new revisions of external libraries, frameworks or new hardware. Perfective changes are applied to implement new or changed (functional or extra-functional) requirements, e.g., changed requirements on quality attributes such as performance (Swanson, 1976; Lientz and Swanson, 1981).

We provide a list of possible changes and affected models in Table 6.1. We consider each of the listed changes to be possible during system operation. For example, in an application server cluster, physical machines can be replaced without system down-time. System changes can be performed either manually, or automatically in the context of an autonomic performance-aware resource management (see Section 1.5). Automatically

Corrective Change	Affected Model
replace component	Application Architecture Deployment
re-compose components	Application Architecture Deployment
Adaptive Change	**Affected Model**
replace component	Application Architecture Deployment
re-compose components	Application Architecture Deployment
replace (parts of) execution environment	Deployment Resource Landscape
replace hardware	Deployment Resource Landscape
Perfective Change	**Affected Model**
replace component	Application Architecture Deployment
re-compose components	Application Architecture Deployment
replace hardware	Deployment Resource Landscape
replace (parts of) execution environment	Deployment Resource Landscape
add/remove hardware	Deployment Resource Landscape
change resource allocation	Deployment

Table 6.1: List of Possible Changes

performed changes can be adaptive changes if the system should be adapted to changing workloads, or perfective changes, if the system should satisfy changing performance objectives such as Service Level Agreements (SLAs).

In case of a change, the affected models have to be maintained. In each of these changes, the model's *structure* needs to be adapted. We denote a model change to be *structural*, if model entities are removed/added/replaced or associations between model entities are altered. Changes of model parameter values are thus not considered as structural changes.

If the system change is triggered automatically as part of an autonomic resource management process, model changes must also be automatically performed. Figure 6.3 shows the implementation components of the control loop introduced with Figure 1.1, cf. Huber (2014). Within the control loop, the *ModelAdaptor* component is responsible for performing the changes on the model level while the *SystemAdaptor* performs the system changes. If the system change is triggered manually, the *ModelAdaptor* can also be used to perform

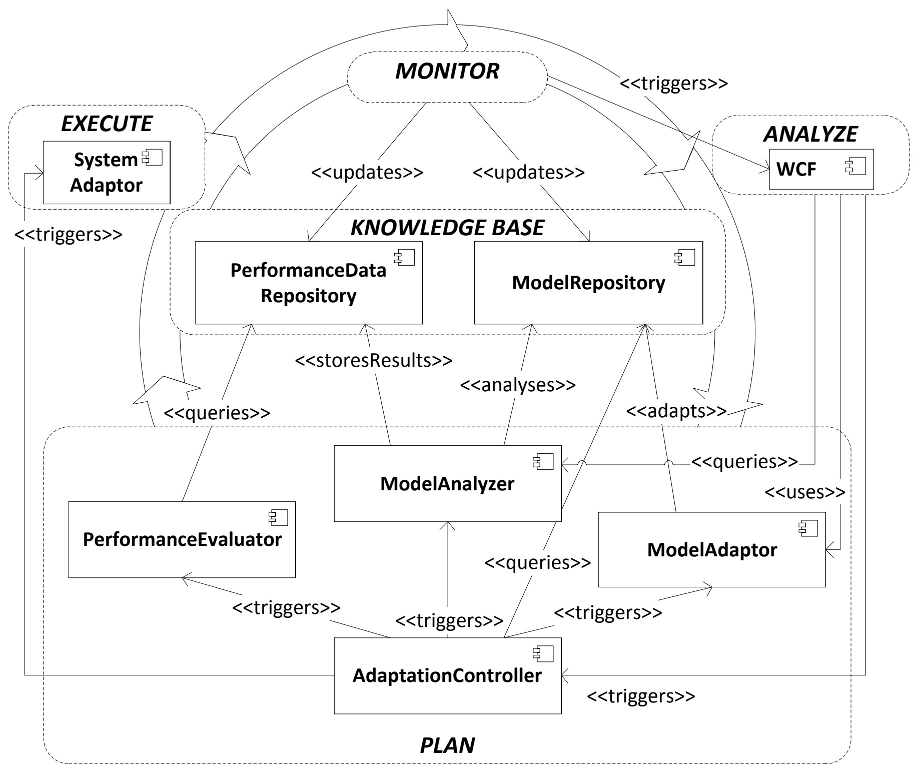


Figure 6.3: Components Constituting Huber's Adaptation Framework (Huber, 2014)

model changes in a convenient way.

6.4 Model Parameter Maintenance

We assume an architecture-level performance model instance whose structure is representative for the modeled system to be available. We propose to observe the model parameter continuously during system operation. On the one hand, model parameters need to be updated if the model structure changes. On the other hand, model parameters need to be updated over time in order to reflect (i) changes due to changing workloads and (ii) changes due to the evolving system state. For instance, in an enterprise software system the data stored in an underlying database system evolves over time. Given that the performance model approximates the database state using empirical observations, these observations need to be updated over time. In summary, the model parameters should be kept up-to-date as the system evolves.

As model parameters we denote instances of model entity ModelVariable, see Figure 4.23. These are control flow variables such as branching probabilities, loop iteration counts, call frequencies as well as resource demands, response times and call parameters. This section describes techniques how these variables can be parameterized using run-time monitoring data. Section 6.4.1 describes such techniques for control flow variables, Section 6.4.2 is about measuring response times, Section 6.4.3 presents approaches to estimate resource demands, and Section 6.4.5 deals with the question of how probabilistic parameter dependencies can be characterized. The presented techniques can be used to implement the monitoring interface described in Section 4.1.6. The first method *getCharacterizationForModelVariable* returns the characterization for a given model variable and a given component instance where the variable resides. The characterization is returned as a random variable. The second method *getCharacterizationForParameterDependency* returns the characterization for a probabilistic parameter dependency. The characterization is also returned as a random variable.

6.4.1 Control Flow Statistics

When obtaining control flow statistics, we are interested in obtaining branching probabilities, loop iteration counts and call frequencies (see Section 4.1.3) from monitoring data. Such parameters can be derived from observed event records.

Let $E = \{e_i = (l_i, t_i, s_i)\}$ be the set of monitored event records and $E/\mathcal{R}$ the corresponding set of equivalence classes for relation $\mathcal{R}$ (see the definitions of event record and $\mathcal{R}$ in Section 6.1.1). An equivalence class $[x]_\mathcal{R} \in E/\mathcal{R}$ is the set of event records belonging to one system request. Thus, we can denote a system request as $[x]_\mathcal{R}$. In the following, we describe how the control flow statistics can be derived from the set of monitored event records E.

Branching Probabilities

There is a branch b with N branch transitions. If the entry of the branch, denoted as $entry(b)$, as well as the entries of branch transitions bt_j, denoted as $entry(bt_j)$, can be monitored, the branching probability pr_j for branch transition bt_j can be derived. The set of event records that represent an entry of branch b is defined as

$$E_b := \{e_i = (l_i, t_i, s_i) \in E \mid l_i = entry(b)\}.$$

The set of event records that represent an entry of branch transition bt_j is defined as

$$E_{bt_j i} := \{e_i = (l_i, t_i, s_i) \in E \mid l_i = entry(bt_j)\}.$$

Then, for $j = 1, \ldots, N$, the branching probabilities are calculated as

$$pr_j = \frac{\#E_{bt_j}}{\#E_b},$$

where symbol # denotes the cardinality of the sets. Since $\sum_{j=1}^{N} \#E_{bt_j} = \#E_b$ holds, the branching probabilities pr_j sum up to 1.0, i.e., $\sum_{j=1}^{N} pr_j = 1.0$.

Loop Iteration Counts

There is a loop lp with loop body lb. If the entry of the loop, denoted as $entry(lp)$, as well as the entry of the loop body, denoted as $entry(lb)$, can be monitored, the average loop iteration count can be derived. Note that we differentiate between entering the loop and entering the loop body. The set of event records that represent an entry of loop l is defined as

$$E_{lp} := \{e_i = (l_i, t_i, s_i) \in E \mid l_i = \mathit{entry}(lp)\}.$$

The set of event records that represent an entry of loop body lb is defined as

$$E_{lb} := \{e_i = (l_i, t_i, s_i) \in E \mid l_i = \mathit{entry}(lb)\}.$$

Then, the average number of loop iteration counts is calculated as $\frac{\#E_{lb}}{\#E_{lp}}$.

However, in order to derive the PMF of the loop iterations that is characterized by the set of loop iteration counts $N := \{n_1, \ldots, n_l\}$ together with their probabilities $P := \{pr_1, \ldots, pr_l\}$, call path tracing needs to be applied. As described in Section 6.1, the equivalence class $[x]_{\mathcal{R}} \in E/\mathcal{R}$ is the set of event records belonging to one system request. To calculate whether request $[x]_{\mathcal{R}}$ enters loop lp or not, a helper function is defined:

$$\mathit{loopentry}([x]_{\mathcal{R}}) := \#\{e_i = (l_i, t_i, s_i) \in [x]_{\mathcal{R}} \mid l_i = \mathit{entry}(lp)\}.$$

Without loss of generality, it is assumed that a system request enters lp at most once. If not, equivalence class $[x]_{\mathcal{R}}$ would need to be further partitioned so that each partition enters the loop at most once, i.e., so that each partition consists of at most one loop entry and loop exit. The helper function $\mathit{loopentry}([x]_{\mathcal{R}})$ thus evaluates to 1 if system request $[x]_{\mathcal{R}}$ enters loop lp, or evaluates to 0 if lp is not entered. To count the number of loop body iterations of $[x]_{\mathcal{R}}$, we calculate:

$$\mathit{loopcount}([x]_{\mathcal{R}}) := \begin{cases} \#\{e_i = (l_i, t_i, s_i) \in [x]_{\mathcal{R}} \mid l_i = \mathit{entry}(lb)\} & \mathit{loopentry}([x]_{\mathcal{R}}) = 1, \\ \mathit{undefined} & \mathit{loopentry}([x]_{\mathcal{R}}) = 0. \end{cases}$$

Note that it is distinguished whether loop lp is entered or not. If the loop is not entered, we set $\mathit{loopcount}([x]_{\mathcal{R}})$ to *undefined.* Thus, system requests that do not access loop lp are ignored. Note that we do not ignore loop iteration counts of 0, which is the case when system request $[x]_{\mathcal{R}}$ enters the loop but not the loop body.

The set of observed loop iteration counts N is then defined by

$$N = \{n \in \mathbb{N}_0 \mid \exists\, [x]_{\mathcal{R}} \in E/\mathcal{R} : \mathit{loopcount}([x]_{\mathcal{R}}) = n\}.$$

We thus set $l = \#N$ and provide the formula for loop iteration count probability pr_i with $i \in \{1, \ldots, l\}$:

$$pr_i = \frac{\#\{[x]_{\mathcal{R}} \in E/\mathcal{R} \mid \mathit{loopcount}([x]_{\mathcal{R}}) = n_i\}}{\#\{[x]_{\mathcal{R}} \in E/\mathcal{R} \mid \mathit{loopentry}([x]_{\mathcal{R}}) = 1\}} .$$

Sets N and P define a PMF that characterizes the distribution of loop iteration counts of loop lp. If l is large, e.g., $l > 10$, histograms can be used to approximate the loop iteration count distribution, see Section 6.1.3.

Call Frequencies

For a coarse-grained service behavior s, the call frequency of external call ec is obtained in a similar way to the distribution of loop iteration counts. The entry of service s, denoted as $\mathit{entry}(s)$, corresponds to the entry of a loop. The entry of external call ec, denoted as $\mathit{entry}(ec)$, corresponds to the entry of a loop body. If $\mathit{entry}(s)$ and $\mathit{entry}(ec)$ can be monitored, the call frequency of ec can be derived.

The set of event records that represent an entry of service s is defined as

$$E_s := \{e_i = (l_i, t_i, s_i) \in E \mid l_i = \mathit{entry}(s)\}.$$

The set of event records that represent an entry of external call ec is defined as

$$E_{ec} := \{e_i = (l_i, t_i, s_i) \in E \mid l_i = \mathit{entry}(ec)\}.$$

Then, per service s, the average number of external calls ec, is calculated as $\frac{\#E_{ec}}{\#E_s}$.

However, if the set of different counts of external call ec per service s, denoted as $N := \{n_1, \dots, n_l\}$, together with their probabilities $P := \{pr_1, \dots, pr_l\}$ has to be calculated, call path tracing needs to be applied. As described in Section 6.1, the equivalence class $[x]_\mathcal{R} \in E/\mathcal{R}$ is the set of event records belonging to one system request. To calculate whether $[x]_\mathcal{R}$ enters the service s or not, a helper function is defined:

$$\mathit{serviceentry}([x]_\mathcal{R}) := \#\{e_i = (l_i, t_i, s_i) \in [x]_\mathcal{R} \mid l_i = \mathit{entry}(s)\}.$$

Without loss of generality, it is assumed that a system request enters service s at most once. If not, equivalence class $[x]_\mathcal{R}$ would need to be further partitioned so that each partition enters service s at most once, i.e., so that each partition consists of at most one service entry and service exit. The helper function $\mathit{serviceentry}([x]_\mathcal{R})$ thus evaluates to 1 if system request $[x]_\mathcal{R}$ enters service s or evaluates to 0 if s is not entered. To count the number of external calls ec for $[x]_\mathcal{R}$, we calculate:

$$\mathit{callcount}([x]_\mathcal{R}) := \begin{cases} \#\{e_i = (l_i, t_i, s_i) \in [x]_\mathcal{R} \mid l_i = \mathit{entry}(ec)\} & \mathit{serviceentry}([x]_\mathcal{R}) = 1, \\ \mathit{undefined} & \mathit{serviceentry}([x]_\mathcal{R}) = 0. \end{cases}$$

Note that it is distinguished whether service s is entered or not. If the service is not entered, we set $\mathit{callcount}([x]_\mathcal{R})$ to $\mathit{undefined}$. Thus, system requests that do not access s are ignored. Note that external call counts of 0 are not ignored, which is the case when system request $[x]_\mathcal{R}$ enters s but not external call ec.

Set N then is defined by

$$N = \{n \in \mathbb{N}_0 \mid \exists\, [x]_\mathcal{R} \in E/\mathcal{R} : \mathit{callcount}([x]_\mathcal{R}) = n\}.$$

We thus set $l = \#N$ and provide the formula for external call count probability pr_i with $i \in \{1, \dots, l\}$:

$$pr_i = \frac{\#\{[x]_\mathcal{R} \in E/\mathcal{R} \mid \mathit{callcount}([x]_\mathcal{R}) = n_i\}}{\#\{[x]_\mathcal{R} \in E/\mathcal{R} \mid \mathit{serviceentry}([x]_\mathcal{R}) = 1\}}\ .$$

Sets N and P define a PMF that characterizes the call frequency of external call ec per service call s. Again, if l is large, e.g., $l > 10$, histograms can be used to approximate the call frequency, see Section 6.1.3.

6.4.2 Response Times

For a black-box service behavior s, the service response time can be measured using event records if the entry of the service call, denoted as $\mathit{entry}(s)$, and the exit of the service call, denoted as $\mathit{exit}(s)$, can be monitored. Since we need to identify pairs of entries and exits, call path tracing needs to be applied.

Let $E = \{e_i = (l_i, t_i, s_i)\}$ be the set of monitored event records and $E/\mathcal{R}$ the corresponding set of equivalence classes for relation $\mathcal{R}$. As described in Section 6.1.1, the equivalence class $[x]_\mathcal{R} \in E/\mathcal{R}$ is the set of event records belonging to one system request. Without loss of generality, it is assumed that a system request calls service s at most once. If not, equivalence class $[x]_\mathcal{R}$ would need to be further partitioned so that each partition calls service s at most once, i.e., each partition consists of at most one service entry and service exit.

To determine whether $[x]_{\mathcal{R}}$ calls service s or not, two helper functions are defined:

$$\begin{aligned} serviceentry([x]_{\mathcal{R}}) :=& \quad \#\{e_i = (l_i, t_i, s_i) \in [x]_{\mathcal{R}} \mid l_i = entry(s)\}, \\ serviceexit([x]_{\mathcal{R}}) :=& \quad \#\{e_i = (l_i, t_i, s_i) \in [x]_{\mathcal{R}} \mid l_i = exit(s)\}. \end{aligned}$$

The helper function *serviceentry* evaluates to 1 if the request calls service s and 0, otherwise. Furthermore, for each request $[x]_{\mathcal{R}}$, $serviceentry([x]_{\mathcal{R}}) = serviceexit([x]_{\mathcal{R}})$ holds.

For an individual request $[x]_{\mathcal{R}}$, the response time is then calculated as

$$responsetime([x]_{\mathcal{R}}) := \begin{cases} t_j - t_i & serviceentry([x]_{\mathcal{R}}) = 1, \\ undefined & serviceentry([x]_{\mathcal{R}}) = 0, \end{cases}$$

where $e_i = (l_i, t_i, s_i) \in [x]_{\mathcal{R}} : l_i = entry(s)$ and $e_j = (l_j, t_j, s_j) \in [x]_{\mathcal{R}} : l_j = exit(s)$.

Calculating the response time for each $[x]_{\mathcal{R}} \in E/\mathcal{R}$ then leads to an empirical distribution of the response time of service s. This empirical distribution consisting of a set of measurement samples then needs to be approximated as described in Section 6.1.3.

6.4.3 Resource Demand Estimation

Both coarse-grained service behavior models as well as fine-grained service behavior models allow characterizing a service's resource demanding behavior. To characterize the resource demanding behavior of a component's provided service, the resource demands of its internal computations need to be quantified. Determining resource demands involves identification of the resource type (e.g., CPU, HDD I/O, Network I/O) used by the service and quantification of the amount of time spent using these resources. The resource demand of a service is its total processing time at the considered resource not including any time spent waiting for the resource to be made available (see also Section 2.2).

Resource demands typically cannot be directly measured, they are estimated based on measurements of other metrics. Resource demand estimation is an established research area. Typically, resource demands are estimated based on measured response times or resource utilization and throughput data (Menascé et al., 1994; Rolia and Vetland, 1995). Most approaches focus on CPU and I/O resources, i.e., memory accesses are normally not considered explicitly. We provide an overview of existing techniques to resource demand estimation based on the master's thesis of Spinner (2011) that has been supervised by the author. Each technique comes with its own advantages and disadvantages in terms of accuracy, robustness and applicability. For instance, there are notable differences in the type and amount of measurement data that is required as method input. The overview is concluded by a classification categorizing the existing techniques to support the decision which technique to apply in which context.

We use the notation presented in Table 6.2 for the description of the different approaches. Note that in order to conform with the notation that is established in the area of resource demand estimation, in the remainder of this section we refer to resource demands of *workload classes* instead of resource demands of services. In the following, resources are denoted with the index i and workload classes are denoted with the index c.

Furthermore, the *Flow Equilibrium Assumption* (Menasce and Virgilio, 2000) is assumed to hold, i.e., that over a sufficiently long period of time the number of request arrivals equals to the number of request completions. As a result, the arrival rate λ_c is assumed to be equal to the throughput X_c.

$D_{i,c}$	average resource demand of requests of workload class c at resource i
$U_{i,c}$	average utilization of resource i due to requests of workload class c
U_i	average total utilization of resource i
λ_c	average arrival rate of workload class c
X_c	average throughput of workload class c
R_c	average response time of workload class c
I	total number of resources
C	total number of workload classes

Table 6.2: Metrics for Resource Demand Estimation Techniques

Approximation with Response Times

The measured response time of a request at a resource is the sum of the waiting time and the resource demand. If one can assume that the waiting time is insignificant compared to the resource demand and the measured response time does not include significant time spent at other resources, the measured response time can be used to approximate the resource demand (Nou et al., 2009; Urgaonkar et al., 2007; Brosig et al., 2009).

This approximation allows the estimation of average resource demands as well as the estimation of the empirical distribution of resource demands. The latter is particularly helpful if the actual resource demand has a large variance and/or has a multi-modal distribution.

Service Demand Law

The utilization at resource i due to requests of workload class c can be derived using the Utilization Law (e.g., Menascé et al. (1994)). Solving for the resource demand leads to the Service Demand Law (e.g., Menascé et al. (1994)):

$$D_{i,c} = \frac{U_{i,c}}{X_{i,c}} \ .$$

This relationship can be used to determine average resource demands based on measured utilization and throughput data. In cases where requests of different workload classes arrive at the system of interest simultaneously, a measured total utilization U_i needs to be apportioned appropriately among the different workload classes. This can be done by ratios obtained from additional per-class metrics provided by the operating system (Lazowska et al., 1984; Menascé et al., 2004b) or from workload class response times (Brosig et al., 2009).

Linear Regression

A common way to infer resource demands is based on linear regression (Bard and Shatzoff, 1978; Rolia and Vetland, 1995; Pacifici et al., 2008; Casale et al., 2007; Zhang et al., 2007; Kelly and Zhang, 2006; Stewart et al., 2007). Given a workload consisting of multiple workload classes, the linear model is usually defined based on the Utilization Law:

$$U_i^{(j)} = \sum_{c=1}^{C} \lambda_c^{(j)} D_{i,c} \ ,$$

where index (j) denotes measurement samples obtained in time window j. For the regression to be meaningful, we need to obtain at least N simultaneous measurement samples, where N is the number of resource demands to estimate.

Commonly, non-negative Least Squares (LSQ) regression is used to solve the model (Rolia and Vetland, 1995; Pacifici et al., 2008). However, the following issues can arise: i) resource demands are stochastically distributed, thus estimating mean resource demands $D_{i,c}$ may lead to significant estimation errors (Rolia and Vetland, 1995); ii) close correlations between the control variables (*multicollinearity*) may cause non-unique and unstable solutions and therefore should be avoided (Pacifici et al., 2008). Ad-hoc techniques to reduce the influence of multicollinearity are presented in Pacifici et al. (2008). Further techniques increasing the robustness of the regression to cope with multicollinearity, outliers and discontinuities due to software or hardware upgrades include Least Absolute Differences (LAD) regression (Stewart et al., 2007) or Least Trimmed Squares (LTS) regression (Casale et al., 2007, 2008).

Kalman Filter

A Kalman filter estimates the hidden state of a dynamic system (Simon, 2006). The authors in Zheng et al. (2008); Kumar et al. (2009a); Wang et al. (2012) apply it for resource demand estimation. The following filter description is based on Kumar et al. (2009a). The system state vector is defined as:

$$\mathbf{x} = \begin{pmatrix} D_1 & \cdots & D_C \end{pmatrix}^T .$$

Without any a-priori knowledge about the system state dynamics, the system state model that describes how the system state evolves over time is reduced to

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{w}_k,$$

where index k denotes discrete time steps. A process noise term $\mathbf{w}_k$ is assumed to be normally distributed with zero mean. The relationship between system state $\mathbf{x}_k$ and measurements $\mathbf{z}_k$ at time step k is denoted as measurement model. If the resource can be represented by a M/M/1 queue (for Kendall's notation describing queues, see Kendall (1953)), the measurement equation can be described by:

$$\mathbf{z} = h(\mathbf{x}) = \begin{pmatrix} R_1 \\ \cdots \\ R_C \\ U \end{pmatrix} = \begin{pmatrix} \frac{D_1}{1-U} \\ \cdots \\ \frac{D_C}{1-U} \\ \Sigma_{c=1}^{C} \lambda_c D_c \end{pmatrix} .$$

The measurement equation is of non-linear nature. To derive a linear measurement model for the measurements $\mathbf{z}_k$, the extended Kalman filter design (Simon, 2006; Kumar et al., 2009a) can be used:

$$\mathbf{z}_k = \mathbf{H}\mathbf{x}_k + \mathbf{v}_k, \text{where } \mathbf{H} = \frac{\partial h}{\partial \mathbf{x}},$$

where $\mathbf{v}_k$ is the observation noise, which is assumed to be white gaussian noise with zero mean. In Zheng et al. (2005, 2008), the authors give recommendations on how to choose filter configurations such as initial state vectors or covariance matrices of process and observation noise.

Optimization

In this paragraph, we describe estimation approaches that are defined as optimization problems and solved with mathematical programming methods. In contrast to the presented linear regression approaches, the estimation approaches described here are based on more general objective functions.

In Liu et al. (2003); Wynter et al. (2004); Liu et al. (2006); Kumar et al. (2009b), the objective function aims at reducing the prediction error of response times and resource utilization:

$$\min \left(\sum_{c=1}^{C} p_c (R_c - \tilde{R}_c)^2 + \sum_{i=1}^{I} (U_i - \tilde{U}_i)^2 \right),$$

where $\tilde{R}_c$ denotes the measured response time of workload class c and $\tilde{U}_i$ the measured utilization of resource i. Expressions of R_c and U_i are derived from standard queueing formulas (Menascé et al., 1994; Bolch et al., 1998). The factor p_c weights the response time errors, e.g., with the proportion of the number of requests of workload class c, $p_c = \frac{\lambda_c}{\sum_{d=1}^{C} \lambda_d}$.

The authors in Kumar et al. (2009b) describe an optimization approach that supports the estimation of load-dependent resource demands, requiring a-priori knowledge of the type of function, e.g., polynomial, exponential or logarithmic, that best describes the relation between workload and resource demand. An approach using series of experiments to increase robustness to noisy measurements and outliers is described in Liu et al. (2006). The optimization does not only consider the current optimal solution but the set of all optimal solutions since the first experiment.

The work in Menascé (2008) formulates an optimization problem that depends only on response time and arrival rate measurements:

$$\min \sum_{c=1}^{C} (R_c - \tilde{R}_c)^2 \text{ with } R_c = \sum_{i=1}^{I} \frac{D_{i,c}}{1 - \sum_{d=1}^{C} \lambda_d D_{i,d}}$$

$$\text{subject to } D_{i,c} \geq 0 \quad \forall i, c \text{ and } \sum_{c=1}^{C} \lambda_c D_{i,c} < 1 \quad \forall i.$$

Other Approaches

Kraft et al. (2009) use Maximum Likelihood Estimation (MLE) to estimate resource demands. MLE allows infering statistics of a random variable by determining the probability of observing a certain sample path. The authors use MLE with measured response times $R_i^1, \ldots, R_i^N$ and queue lengths that were seen on arrival of a request. They then search for the resource demands $D_{i,1}, \ldots, D_{i,C}$ so that the probability of observing the measured response times is maximized.

Rolia et al. (2010a) propose a technique for estimating the aggregate resource demand of a workload mix, called Demand Estimation with Confidence (DEC). This technique assumes that a set of benchmarks utilizing different functions of an application is available. Based on measured demands for the individual benchmarks, DEC can estimate the aggregate resource demand of a given workload mix.

Kalbasi et al. (2011) consider the use of Support Vector Machines (SVMs) (Smola and Schölkopf, 2004) for estimating resource demands. They compare it with results from LSQ and LAD regression and show that it can provide better resource demand estimates depending on the characteristics of the workload.

Classification

After having presented existing approaches to resource demand estimation with their advantages and disadvantages, the list is summarized in Table 6.3. The most common techniques use response time approximation, the Service Demand Law, or linear regression based on the Utilization Law. Further techniques apply Kalman filtering, formulate general optimization problems, or apply MLE.

Class	**Variant**	**Approach (identified by authors)**
Response time approximation	Single resource	Brosig et al. (2009)
	Multiple resources	Nou et al. (2009) Urgaonkar et al. (2007)
Service Demand Law		Brosig et al. (2009) Lazowska et al. (1984)
Linear regression	LSQ	Rolia and Vetland (1995, 1998) Pacifici et al. (2008) Kraft et al. (2009)
	LAD	Zhang et al. (2007); Stewart et al. (2007)
	Robust regression	Casale et al. (2007, 2008)
Kalman filter	Single workload class	Zheng et al. (2005, 2008)
	Multiple workload classes	Kumar et al. (2009a)
Optimization		Menascé (2008) Zhang et al. (2002)
	Inferencing	Liu et al. (2003); Wynter et al. (2004); Liu et al. (2006)
	Enhanced Inferencing	Kumar et al. (2009b)
MLE		Kraft et al. (2009); Perez et al. (2013)
DEC		Rolia et al. (2010a,b)

Table 6.3: List of Approaches to Resource Demand Estimation, cf. Spinner (2011)

We showed that there are notable differences in the type and amount of measurement data that is required as input for the approaches. Table 6.4 provides an overview. Parameters common to all estimation approaches, such as the number of workload classes and the number of resources, are not included. Most linear regression approaches are based on the Utilization Law. They require the total utilization of a resource and the throughput of each workload class. In contrast, the linear regression approach described by Kraft et al. (2009) depends on the measured response time and the queue length seen upon arrival. For the Kalman filter, varying definitions are in use. Zheng et al. (2008) propose to use a subset of: throughput, end-to-end response time, total delay at a resource, total utilization of a resource or the mean number of requests at a resource. The DEC approach (Rolia et al., 2010a) is based on per-class throughput and per-resource visit counts. Additionally, it is assumed that a set of benchmarks stressing different parts of the system is available. Some of the estimation approaches also depend partly on resource demands that are known beforehand. Menascé (2008) suggests how to calculate missing resource demands from a set of given resource demands. The given resource demands can come from other estimation approaches. Another approach that requires partial resource demands for each workload class is described by Lazowska et al. (1984). It is assumed that the resource demands are measured with an accounting monitor. Such an accounting monitor, however, does not include the system overhead caused by a workload class. The system overhead is defined as the work done by the operating system when processing a request. Lazowska et al. describes a way to distribute unattributed computing time between the workload classes. Thus, we can obtain improved resource demand estimates that take into account the system overhead. Some of the optimization approaches require information about the scheduling strategy of the involved resources. The description of the optimization problems used by the estimation approaches of Zhang et al. (2002), Liu et al. (2003); Wynter et al. (2004); Liu et al. (2006) and Kumar et al. (2009a) depend on the scheduling strategies of a corresponding queueing model. In addition to the types of input parameters required by an estimation approach, some approaches also provide a rule of thumb regarding the number of required measurement samples. Approaches based on linear regression (Rolia and Vetland, 1995, 1998; Pacifici et al., 2008) need at least $K + 1$ linear independent equations to estimate K resource demands. When using robust regression methods, significantly more measurements might be necessary (Casale et al., 2007). In Kumar et al. (2009a), the authors give a formula to calculate the number of measurements required by their optimization-based approach. Yet, these are only minimum bounds for the number of measurements. A lot more measurements are typically required to obtain good estimates (Stewart et al., 2007).

The approaches to resource demand estimation are typically used to determine the *mean* resource demand of requests of a workload class at a resource. However, resource demands cannot be assumed to be deterministic (Rolia et al., 2010a), e.g., they typically depend on the data processed by an application or on the current state of the system or application (Rolia and Vetland, 1995). In certain scenarios, e.g., if Dynamic Voltage and Frequency Scaling (DVFS) or hyperthreading techniques are activated (Kumar et al., 2009b), resource demands may be load-dependent. In such cases, the mean resource demands are not constant, but a function that may depend, e.g., on the arrival rates of the workload classes (Kumar et al., 2009b).

Thus, the estimated mean value may not be sufficient in some situations. More information about the confidence of estimates and the distribution of the resource demands can be valuable. Estimates of higher moments of the resource demand can also be useful to determine the shape of their distribution. We distinguish between point and interval estimators of the real resource demands. Confidence intervals help to assess the reliability of resource demand estimates, if the underlying statistical model is valid.

Point estimates of the mean resource demand are provided by all considered approaches. Confidence intervals can be determined with the approaches based on response time approximation, linear regression (Rolia and Vetland, 1995; Kraft et al., 2009) and with the DEC approach (Rolia et al., 2010a). The MLE approach (Kraft et al., 2009) and the optimization approach described by Zhang et al. (2002) are also capable of providing estimates of higher moments. This additional information comes at the cost of a higher number of required measurements. Furthermore, all of the presented estimation approaches are capable of estimating load-independent mean resource demands. Additionally, the Enhanced Inferencing approach (Kumar et al., 2009b) also supports the estimation of load-dependent resource demands, assuming a given type of function.

6.4.4 Resource Demand Estimation in Virtualized Environments

The techniques presented in Section 6.4.3 are independent of the type of execution environment. However, in virtualized execution environments, the virtualization layer makes the estimation of resource demands difficult and inaccurate (Brosig et al., 2013a). Most of the mentioned approaches require the measured total resource utilization as input (see

Estimation approach	**Measurements**						**Others**	
	Utilization	Response time	Throughput	Arrival rate	Queue length	Visit counts	Resource demands	Scheduling strategy
Response time approx.								
- Brosig et al. (2009)		✗						
- Urgaonkar et al. (2007)		✗[1]				✗		
- Nou et al. (2009)	✗	✗						
Service Demand Law								
- Brosig et al. (2009)	✗	✗				✗		
- Lazowska et al. (1984)			✗				✗[2]	
Linear regression								
- Kraft et al. (2009)		✗			✗			
- Rolia and Vetland (1995, 1998)	✗		✗					
Kalman filter								
- Zheng et al. (2005, 2008)	✗	✗						
- Kumar et al. (2009a)	✗	✗		✗				
Optimization								
- Menascé (2008)		✗					✗[3]	
- Zhang et al. (2002)	✗		✗					✗
- Inferencing (e.g., Liu et al. (2003))	✗	✗		✗		✗		✗
- Enhanced Inferencing (Kumar et al., 2009b)	✗	✗		✗				✗
MLE (e.g., Kraft et al. (2009))		✗			✗			✗
DEC (Rolia et al. (2010a,b))			✗			✗		

[1] Response time per resource.
[2] Measured with accounting monitor. System overhead is not included.
[3] A selected set of resource demands is known a priori.

Table 6.4: Input Measurements of Estimation Approaches, cf. Spinner (2011)

Table 6.4). However, given that in virtualized environments resources are shared, this metric is often not available or inaccurate (Wood et al., 2008). Thus, the existing techniques neglect the virtualization overhead.

In Section 6.4.4.1, we provide an example to illustrate the inaccuracy and show that the virtualization overhead has a significant performance influence. Section 6.4.4.2 outlines the problem of quantifying the virtualization overhead. Section 6.4.4.3 and Section 6.4.4.4 describe approaches to obtain the virtualization overhead, depending on the type and amount of available monitoring data.

6.4.4.1 Example

Table 6.5 shows an example illustrating the influence of the virtualization overhead on application-level performance metrics from Brosig et al. (2013a). We compare the performance of an operation *CreateVehicleEJB* of the SPECjEnterprise2010[1] benchmark in two

[1]SPECjEnterprise2010 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/jEnterprise2010`.

Throughput	Native AppServer		Virtualized AppServer	
X	$U_{AppServer}$	R_{Avg}	$U_{AppServerVM}$	R_{Avg}
35	14.9%	26ms	15.8%	32ms
65	24.8%	27ms	27.2%	39ms
100	35.7%	28ms	40.0%	48ms
154	53.2%	31ms	60.7%	100ms

Table 6.5: Example: Response Times in Native versus Virtualized Setup

different deployment scenarios. In the first scenario, the benchmark is deployed in a native application server without use of virtualization. In the second scenario, the benchmark is deployed in a Xen-virtualized[2] application server on an identical physical machine with one CPU core. We investigate the operation's average response times and the utilization of the application server under four different load conditions. Throughput X is varied so that the application server CPU has utilization $U_{AppServer}$ in the range of 15% to 60%. The application server Virtual Machine (VM) is the only guest VM hosted by the hypervisor. In the four load scenarios, the utilization of the application server ($U_{AppServer}$) does not vary significantly. The virtualized application server ($U_{AppServerVM}$) has a slightly higher utilization than the native one. The average response times denoted as R_{Avg}, however, are significantly higher in the virtualized setup. Looking at the growth rate of the response times in the virtualized setup leads to the assumption that the virtualized application server is already severely utilized, although the VM shows a utilization of only 61%. Obviously, the virtualization overhead plays an important role when investigating performance properties in virtualized systems.

The deployed Xen hypervisor (version XenServer 5.5) is an open source bare-metal hypervisor (type-I). With the Xen hypervisor, multiple para-virtualized or full-virtualized virtual machines (guest domains) can be executed on a single server sharing the physical resources. A scheduler, integrated in the hypervisor, schedules the access of all domains to the available physical CPUs. For access to other devices and for managing the guest domains, Xen uses a privileged control domain (denoted as Domain-0). Domain-0 contains the device drivers to access the physical devices. All communication of the guest domains with the physical devices goes through Domain-0. This causes additional management overhead in terms of CPU consumption. For example, if a guest domain sends a disk I/O request, Domain-0 requires CPU time to process the request on behalf of the guest domain. Although other virtualization solutions such as VMware ESX[3] implement a different architecture, the problem of additional overhead in the virtualization layer remains.

To obtain more detailed monitoring data from the Xen-virtualized system, we use Xenmon (Gupta et al., 2005; Wood et al., 2008). Xenmon collects monitoring data both from the hosted guest VMs and from Domain-0. Reported metrics include CPU utilization, network traffic and disk I/O accesses. Xenmon reports resource usage (CPU, network I/O, disk I/O) of physical resources as well as resource usage at the virtualization layer issued by guest VMs and Domain-0. When it comes to quantifying the virtualization overhead, the resource utilization attributed to Domain-0 becomes of interest. The monitoring data in Table 6.6 shows the same scenario as in Table 6.5 with added information on the measured utilization accounted to Domain-0, reported by Xenmon. The utilization of Domain-0 ($U_{Domain0}$) ranges from 12% to 31%, depending on the load level. For instance, at the highest load level, the utilization of Domain-0 is half of the utilization measured directly at the application server VM. This explains the highly increased response times

[2]The Xen Project, `http://www.xen.org/`
[3]VMware, `http://www.vmware.com/`

Throughput	Virtualized AppServer		
X	$U_{Domain0}$	$U_{AppServerVM}$	R_{Avg}
35	11.6%	15.8%	32ms
65	18.9%	27.2%	39ms
100	25.2%	40.0%	48ms
154	30.6%	60.7%	100ms

Table 6.6: Example: VM Utilization versus Domain-0 Utilization

of `CreateVehicleEJB`. The system is already heavily utilized, i.e., $> 90\%$. Recall that the application server VM is the only running guest VM in that scenario and thus the only one responsible for the observed Domain-0 load.

6.4.4.2 Problem Formulation

In terms of Xen, it is unclear to what extent each guest VM causes load in Domain-0. The pie chart in Figure 6.4 illustrates the resource utilization of a Xen-virtualized physical resource. The utilization is divided among three guest VMs ($VM_1 - VM_3$) *and* Domain-0. The utilization accounted to Domain-0 again has to be partitioned and accounted to $VM_1 - VM_3$. When considering the physical resource utilization caused by a specific VM, the corresponding Domain-0 partition has to be added to the utilization obtained at the VM level. Note that the illustration does not consider the system overhead, which is regarded as noise that should be equally distributed among all partitions.

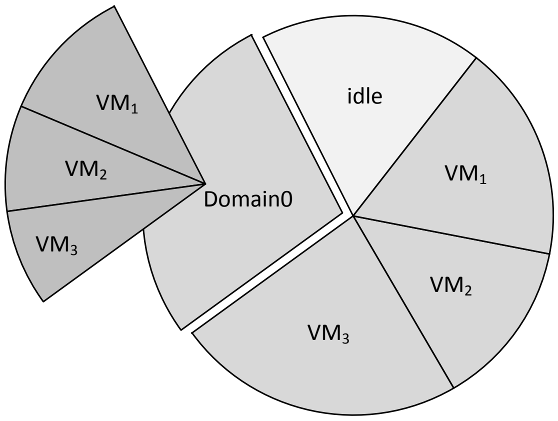


Figure 6.4: Resource Utilization of a Virtualized Resource

More general, assume a hypervisor is running on a physical machine hosting n virtual machines $VM_1, \ldots, VM_n$, where VM_i processes workload classes $w_{i,j}, j \in \{1, \ldots, k_i\}$. The virtual machines and the hypervisor share the available physical resources. As shown in Huber et al. (2011b); Lu et al. (2011), with modern virtualization solutions, CPU-intensive workloads inside the VMs have a negligible overhead of lower than one percent. Network and disk I/O traffic of the VMs, however, induce significantly higher processing overhead. The overhead furthermore depends on the configuration of each hosted VM such as the number of assigned virtual CPUs or parameters such as core affinity and shares (Huber et al., 2011b).

Metric of interest is the resource utilization of the physical and virtual machines. The CPU utilization of the physical machine is denoted as U_{phys}. The CPU utilization of VM_i is denoted as U_{VM_i} and refers to the utilization as measured inside the VM, i.e., the incurred virtualization overhead is not included. The question is how the virtualization overhead of workload class $w_{i,j}$ can be obtained.

$D_{w_{i,j}}$	Mean resource demand of workload class $w_{i,j}$ *without* the virtualization overhead.
$O_{w_{i,j}}$	Virtualization overhead factor for workload class $w_{i,j}$.
$X_{w_{i,j}}$	Throughput of workload class $w_{i,j}$.

Table 6.7: Parameterizing Virtualization Overhead

In the next sections, different ways are described to obtain the virtualization overhead, depending on the type and amount of available monitoring data. Table 6.7 shows the parameters that are relevant in the following. The resource demands $D_{w_{i,j}}$ indicate the mean resource demand *without* the virtualization overhead. They can be derived using existing techniques as presented in Section 6.4.3. The challenge is to estimate the virtualization overhead factors $O_{w_{i,j}}$. There are different ways to obtain the relevant overhead, depending on the type and amount of available monitoring data.

6.4.4.3 Global Overhead Factors and Offsets

One option to approximate the virtualization overhead is to make use of a set of benchmarks. The set of benchmarks is then used to characterize the target virtualization infrastructure for different types of workloads such as CPU-bound, memory-intensive, disk I/O- or network I/O-intensive workloads. Selected benchmarks are, e.g., Passmark PerformanceTest[4], SPEC CPU 2006[5], and Iperf[6] (Huber et al., 2011b, 2010). Huber et al. (2011b, 2010) propose an automated experimental analysis approach to derive the benchmark result deltas between a native and a virtualized system. The benchmark result deltas are then used to calculate virtualization overhead factors for the different types of workloads. The virtualization overhead factors $O_{w_{i,j}}$ are chosen according to the most similar workload type corresponding to workload class $w_{i,j}$.

6.4.4.4 Application-Specific Overhead Portions

If the application is available for performance testing, the virtualization overhead factors can be estimated based on a more precise workload profile of the application. First, a set of microbenchmarks is used to parameterize an overhead model for different I/O activities such as disk or network I/O accesses. Then, a resource access profile of the software system is obtained. Applying the overhead model to the application's resource access profile one can estimate the virtualization overhead factors.

The authors in Wood et al. (2008); Gupta et al. (2005) use this approach to estimate the resource requirements of applications when they are migrated from a native to a virtualized environment. They execute microbenchmarks on a machine both in a virtualized and a native deployment. They build a regression model with resource access metrics collected in the native environment together with CPU utilization metrics (for Domain-0 and the VM) collected in the virtualized environment. The result is a model with a resource access profile as input, and CPU utilization for Domain-0 and the VM as output. Motivated by Wood et al. (2008), Lu et al. (2011) identifies the metrics in Table 6.8 as sufficient to parameterize an overhead model for I/O activities in Xen 3.3.1. The Domain-0 CPU overhead issued by network accesses has a "clear linear relationship with the packet sending and receiving rates, but not with the network throughput in bytes" (Lu et al., 2011). The

[4]http://www.passmark.com/products/pt.htm
[5]http://www.spec.org/cpu2006/
[6]http://iperf.sourceforge.net/

Network accesses	Disk accesses
M_1 := Received packets/sec	M_3 := Read blocks/sec
M_2 := Sent packets/sec	M_4 := Write blocks/sec

Table 6.8: Resource Utilization Metrics Characterizing I/O Activities

same applies to the disk read/write block rates, which have a linear relationship to the Domain-0 CPU overhead generated by disk accesses. In contrast to I/O, virtual CPU accesses and memory accesses do not impose a significant CPU overhead on Domain-0 (Lu et al., 2011).

In the following, the work of Lu et al. (2011) and Wood et al. (2008) is combined to an approach to estimate virtualization overhead factors (Brosig et al., 2013a). To measure the overhead, resource-specific microbenchmarks are executed in a single VM on a physical machine collecting statistics about resource accesses and Domain-0 utilization. SysBench (Lu et al., 2011) can be used as a disk I/O microbenchmark and Netperf (Lu et al., 2011) as a network I/O microbenchmark. For different time intervals of microbenchmark executions, equation

$$U_{Dom0} = p_0 + \sum_{l=1}^{4} p_l * M_l \tag{6.1}$$

is formulated. The Domain-0 utilization is thus a linear combination of the above mentioned metrics. To solve the set of equations, established regression techniques such as LSQ regression can be applied. The output of the regression are overhead factors for specific I/O activities. An approximation of the Domain-0 utilization is computed as

$$\hat{U}^{w_{i,j}}_{Dom0} := p_0 + \sum_{l=1}^{4} p_l * M_l^{w_{i,j}}$$

where $M_l^{w_{i,j}}$ are the average access rates of workload class $w_{i,j}$. These average access rates can be obtained together with metric $U^{w_{i,j}}_{phys}$ from a native deployment. Note that the machine should be identical or similar to the machine where the microbenchmarks are executed; for cross-platform overhead models we refer to Wood et al. (2008). The approximate overhead factors are then computed as

$$O_{w_{i,j}} = 1 + \hat{U}^{w_{i,j}}_{Dom0} / U^{w_{i,j}}_{phys}.$$

If the application is running in a virtualized environment, the virtualization overhead can be obtained from Domain-0 utilization measurements:

$$O_{w_{i,j}} = 1 + U^{w_{i,j}}_{Dom0} / U^{w_{i,j}}_{VM_i},$$

where $U^{w_{i,j}}_{VM_i}$ is the measured utilization of VM_i due to $w_{i,j}$ and $U^{w_{i,j}}_{Dom0}$ is the measured Domain-0 utilization partition induced by $w_{i,j}$. However, when running more than one VM, the latter Domain-0 partition has to be approximated.

The authors in Lu et al. (2011) describe a method that partitions the Domain-0 utilization in different blocks, where each block can be assigned to a guest VM that caused the utilization. As input, the method requires VM monitoring data of disk I/O, network I/O and CPU usage for each guest VM as well as for Domain-0. The output is an approximation of the per-VM physical resource utilization. The method uses a regression model like Equation 6.1 as a starting point, and calibrates/adjusts the model using run-time monitoring data of I/O metrics (Table 6.8) as well as resource utilization metrics for Domain-0 and for the guest VMs. The regression model estimation error is continuously observed and triggers an adjustment process when a certain threshold is reached. This way, the model reflects workload dynamics that may be caused by changes in workload patterns. The authors propose a guided regression as an adjustment approach, for details we refer to Lu et al. (2011).

The evaluation scenarios for partitioning CPU utilization presented in Lu et al. (2011) exhibit a relative error of lower than 10%, mostly around 5%. The experimental evaluation in Wood et al. (2008) exhibits comparable error rates: After a model refinement step that includes post-processing the training data (to eliminate or re-run erroneous microbenchmark runs) and rebuilding the regression model, 90% of the overhead estimations for Domain-0 are within 5% accuracy, and within 10% for estimating the VM CPU utilization.

6.4.5 Probabilistic Characterization of Parameter Dependencies

The measurement and estimation of the model parameters described in the previous four sections (Section 6.4.1 to Section 6.4.4) is done independently of other parameters. However, a model parameter of a service behavior can depend on one or more input parameters passed when invoking the service. For instance, a branch probability might heavily depend on the value of an input parameter. In such cases, it is desirable to be able to quantify such dependencies. By monitoring service input parameters and relating observed parameter values with the observed service control flow, probabilistic models of parameter dependencies can be derived.

If no a priori information about the existence of parametric dependencies is available, their automatic discovery based on monitoring data alone is a challenging problem that requires the use of complex correlation analysis or machine learning techniques (Krogmann et al., 2010). An automatic detection method has to cope with high degrees-of-freedom: Each control flow construct might depend on multiple input parameters and the possible dependencies are not restricted to parameter values, other parameter characterizations such as the length of an array might also have an influence. For these reasons, we consider the automatic discovery of parametric dependencies at run-time based on monitoring data alone to be impractical. It is assumed that information about the existence of potential performance-relevant parameter dependencies is available a priori, defined using the modeling abstractions presented in Section 4.1.5.

This section describes how the monitoring method *getCharacterizationForParameterDependency* of the monitoring interface introduced in Section 4.1.6 can be implemented. The method returns the characterization for the dependent variable of a parameter dependency. A parameter dependency is determined by the method's arguments:

- a list of references to ModelVariables $mv_1, \ldots, mv_n$ that are the independent variables,
- a list of Literals $v_1, \ldots, v_n$ that are values for the above-mentioned variables,
- a list of ComponentInstanceReferences indicating where $mv_1, \ldots, mv_n$ reside,
- a reference to a ModelVariable d that is the dependent variable,
- a reference to a component instance indicating in which component instance d resides.

Formally, the parameters specify the signature of a function to be evaluated at arguments $v_1, \ldots, v_n$. The result of the function, returned as a random variable, characterizes the dependent variable. If $n = 0$, then *getCharacterizationForParameterDependency* has the same semantics as *getCharacterizationForModelVariable*.

The evaluation of the function is based on monitoring data. The function is defined by observations of an n-tuple of independent variable values and an observation of a dependent variable value, i.e., the observations serve as *training data* for the function. In the research area of machine learning, this is called Supervised Learning (Mohri et al., 2012). A supervised learning algorithm tries to generalize a function from inputs to outputs, so that the function can return outputs for previously unseen inputs.

However, the method of the monitoring interface is kept generic, but it is not intended to observe each parameter dependency between possible independent and dependent variables. In Section 6.4.5.1, we discuss how to obtain the set of parameter dependencies to be observed. Section 6.4.5.2 describes how observations respectively training data of a parameter dependency are collected. Finally, Section 6.4.5.3 presents appropriate supervised learning algorithms.

6.4.5.1 Parameter Dependencies to Monitor

Performance-relevant parameter dependencies are indicated by Relationships that are defined in Section 4.1.5. Two kinds of Relationships are distinguished: There are DependencyRelationships connecting one or more InfluencingParameters with one InfluencedVariableReference and there are DependencyPropagationRelationships connecting one or more InfluencingParameters with another InfluencingParameter.

In order to obtain the parameter dependencies to monitor, there are several options:

- One option is to monitor each Relationship. A Relationship consists of one or more references to ModelVariables as independent variables, and one reference to a ModelVariable as dependent variable. However, an involved InfluencingParameter can be a ShadowParameter, i.e., an InfluencingParameter that is not observable. If the dependent variable of a Relationship is such a non-observable parameter, the Relationship cannot be monitored and has to be ignored. If one of the independent variables of the Relationship is a non-observable parameter, the Relationship can be monitored ignoring that variable in case it is not the only independent variable.

- Another option is to make use of the fact that the relationship solving algorithm (see Section 5.2) calls the monitoring interface to resolve Relationships and to parameterize the model. When a performance prediction is triggered by a performance query as described in Chapter 5, relationship solving leads to those calls of the monitoring interface that are actually relevant to answer the performance query. Thus, to determine the relevant parameter dependencies, the monitoring infrastructure does not need to search the model instance for relationships itself, it is the relationship solving algorithm that can be used in a *dry run* to obtain the relevant parameter dependencies that need to be observed.

  In the example in Section 5.2.2, the monitoring interface is called multiple times (see Table 5.1). However, the monitoring methods only need to return non-null values if either a resource demand, a response time, or a control flow variable has to be characterized. The other calls to the monitoring interface stem from intermediate steps required to transitively resolve the relationships. Hence, whenever monitoring method *getCharacterizationForParameterDependency* is called and the dependent variable is one of the above-mentioned types, the parameter dependency determined by the independent variables is considered to be monitored.

  A precondition to obtain the relevant parameter dependencies using the relationship solving algorithm in a dry run is that a UsageProfileModel with UsageScenarios is already available. Furthermore, a performance query has to be formulated to trigger the performance prediction and thus the solving algorithm. If a set of possible performance queries is known beforehand, they can be used to obtain the minimal amount of parameter dependencies to monitor. If no queries are known beforehand, an artificial query that requests to return all available metrics can be constructed. This artificial query can then be used to obtain the set of parameter dependencies to monitor.

Irrespective of the chosen option, the result is a set of parameter dependencies to monitor. A parameter dependency is defined by a list of independent variables and one dependent variable, together with a component instance reference indicating where the variables to monitor reside.

6.4.5.2 Training Data

Having identified a parameter dependency to monitor, this section describes how training data for the probabilistic characterization of the dependency is collected.

The independent variables of the parameter dependency are denoted as $mv_1, \dots, mv_n$. The dependent variable is denoted as d. Let $E = \{e_i = (l_i, t_i, s_i)\}$ be the set of monitored event records and $E/\mathcal{R}$ the corresponding set of equivalence classes for relation $\mathcal{R}$ (see the definitions of event record and $\mathcal{R}$ in Section 6.1.1). Call path tracing needs to be applied in order to map observed values of independent variables to a value of the dependent variable. As described in Section 6.1, the equivalence class $[x]_{\mathcal{R}} \in E/\mathcal{R}$ is the set of event records belonging to one system request. In the following, we describe how a tuple $(obs(mv_1), \dots, obs(mv_n), obs(d))$ can be derived from E, where $obs(mv_1), \dots, obs(mv_n)$ are the observed values for $mv_1, \dots, mv_n$ and $obs(d)$ is the corresponding observed value for d. The tuple can then be used as training data.

Independent variables are of type CallParameter, i.e., they are either service input parameters, external call parameters or external call return parameters (see Figure 4.25). Thus, independent variables are attached to entries or exits of service calls respectively external calls. If the entries and exits can be monitored together with the actual parameter values (input parameter values or return parameter values), for a request $[x]_{\mathcal{R}} \in E/\mathcal{R}$ the values $obs(mv_1), \dots, obs(mv_n)$ can be derived and attached to $[x]_{\mathcal{R}}$ as payload.

The dependent variable d is either a set of branching probabilities, a loop iteration count, a call frequency, a response time, a resource demand or another call parameter.

- Branch probability: If d represents the branching probabilities of branch b with branch transitions $bt_1 \dots, bt_N$, then $obs(d) = branchtransition([x]_{\mathcal{R}})$ where

  $$branchtransition([x]_{\mathcal{R}}) := \begin{cases} j & \exists e_i = (l_i, t_i, s_i) \in [x]_{\mathcal{R}} : l_i = entry(bt_j), \\ undefined & \text{otherwise.} \end{cases}$$

  Thus, the observation is the index of the executed branch transition.

- Loop iteration count: If d is the loop iteration count of a loop, then $obs(d) = loopcount([x]_{\mathcal{R}})$ (see Section 6.4.1 for the definition of function *loopcount*).

- Call frequency: If d is the call frequency of an external call, then $obs(d)$ is set to $callcount([x]_{\mathcal{R}})$ (see Section 6.4.1 for the definition of function *callcount*).

- Response time: If d is the response time for a service call, then $obs(d)$ is set to $responsetime([x]_{\mathcal{R}})$ (see Section 6.4.1 for the definition of function *responsetime*).

- Resource demand: If d is a resource demand, it is required to differentiate. If the resource demand is estimated using response time approximation, $obs(d)$ can be set to the response time of the respective call, monitored within request $[x]_{\mathcal{R}}$. However, if the resource demand is estimated using a different approach (see Section 6.4.3), a resource demand sample cannot be obtained from a single observation of $[x]_{\mathcal{R}}$. In that case, a multi-dimensional histogram has to be built using approaches as presented in Bruno et al. (2001); Gunopulos et al. (2000): The n independent variables translate into n dimensions. Each histogram bucket then defines a workload class in terms of resource demand estimation. The resource demands are then estimated for each of these workload classes separately.

- Call parameter: If d is a call parameter whose actual parameter value can be monitored, $obs(d)$ is set to that value.

For a request $[x]_{\mathcal{R}} \in E/\mathcal{R}$, the monitored tuple $(obs(mv_1), \ldots, obs(mv_n), obs(d))$ may be incomplete, meaning that some elements of the tuple are *undefined*. The tuple can be used as training data, if $obs(d)$ as well as at least one of the values $obs(mv_i)$ is defined. Obtaining the tuple for all requests $[x]_{\mathcal{R}} \in E/\mathcal{R}$ then leads to a training data set.

6.4.5.3 Supervised Learning

A supervised learning algorithm tries to generalize a function from inputs to outputs, so that the function can return outputs for previously unseen inputs (Mohri et al., 2012). In others words, the function serves as predictive model that maps observed inputs to a target output.

In the literature there exist already many such techniques, applied in research areas such as data mining and machine learning. The predictive model is typically represented by a decision tree. Common names for decision trees are also classification trees or regression trees (Breiman et al., 1984). The difference between a classification tree is the domain of the target output. The outcomes of a classification tree analysis are of a discrete finite domain. Regression tree analysis is applied if the predicted outcome can be considered as a continuous number.

In the following, we describe an example to illustrate a decision tree analysis. Figure 6.5 shows a data set of 200 training data tuples for which a decision tree should be build. There are two input variables x_1 and x_2, each with values between 0 and 100. The output variable has two possible outcomes, either '`x`' or '`o`'. Figure 6.6 shows the partitions on the

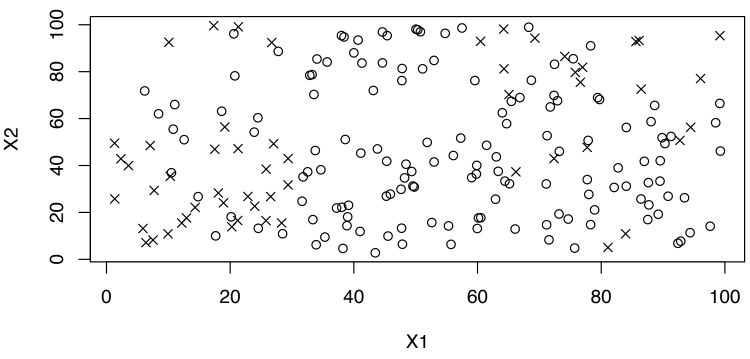


Figure 6.5: Example: Data Set for Classification Tree

data set as they are built by binary recursive partitioning (Breiman et al., 1984). Note that the R statistics tool[7] provides implementations of common decision tree analysis

[7]The R Project, `http://www.r-project.org/`.

methods. In this simple example, there are five partitions. Each partition is labelled with a probability of observing outcome 'x' for a given input of x_1 and x_2. The probability is just the sample mean of observing 'x' in the partition. For instance, in the lower left partition as well as in the upper right partition, the probability of observing outcome 'x' is higher than the probability of observing outcome 'o'. Figure 6.7 shows the actual decision

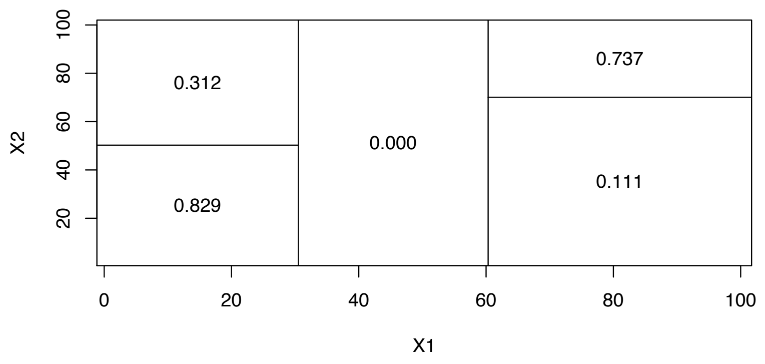


Figure 6.6: Example: Partitions

tree that corresponds to the partitions shown in Figure 6.6. Each tree leaf represents one partition.

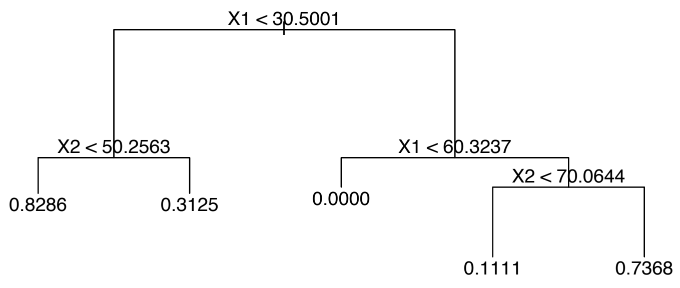


Figure 6.7: Example: Classification Tree

As presented in the previous Section 6.4.5.2, in our context, the training data set consists of tuples of the form $(obs(mv_1), \ldots, obs(mv_n), obs(d))$. The target output is either a branch transition index, a loop iteration count, a call frequency, a response time, or a resource demand. Note that the branch transition index is the only target domain that is finite and discrete. The domains of the input variables depend on the datatype of the model variables mv_i, i.e., the datatype of the CallParameter mv_i represents. Applying Classification and Regression Tree (CART) analysis (Breiman et al., 1984) on the training data set allows predicting a *constant* estimate given observations $(obs(mv_1), \ldots, obs(mv_n))$. Using CART predictive models have the following advantages:

- A decision tree analysis uses both (i) locality within the input space as well as (ii) locality within the output space to build the partitions. Two tuples of the training data set are likely to end in the same partition, if (i) the input variables $(obs(mv_1), \ldots, obs(mv_n))$ are similar and (ii) the outcome is similar.

- There are established algorithms available to learn a tree. The algorithms are highly configurable to influence tree building. If a tree becomes too complex because of "over-training", there are also pruning algorithms to simplify the tree (Breiman et al., 1984; Hastie et al., 2001).
- If some input variables are missing, it is still possible to obtain a prediction. One might not be able to go all the way down a decision tree to a leaf, but one can still obtain a prediction by averaging the leaves in the reachable sub-tree.

However, CART predictive models only provide a single value. Depending on the type of variable d, CART methods allow the prediction of a single branch transition, loop iteration count, call frequency, response time or resource demand. The characterizations of interest, however, are random variables respectively their probability distributions (see Section 4.1.4). For instance, instead of returning the most probable branch transition, the outcome should be a PMF with the set of possible branch transitions as domain. Similarly, instead of returning a loop iteration count, the outcome should be a loop iteration count *distribution* characterized as a PMF over the observed loop iteration counts. A leaf in the decision tree should thus not be only represented by the sample mean of the observed outcome in the partition that corresponds to the leaf. Instead, a partition should be characterized by an empirical distribution derived from the observed outcomes in the partition. For this purpose, one can use the same aggregation methods as they are presented in Section 6.1.3. If a partition is characterized using a PMF, the decision tree can be kept simple, i.e., the number of partitions can be small.

6.5 Model Calibration and Adjustment

In capacity planning, prediction errors of up to 30% concerning the average response time and 5% concerning the resource utilization are considered acceptable (Menascé et al., 1994; Menasce and Virgilio, 2000). If a performance model does not provide such an acceptable accuracy, the performance model needs to be calibrated and adjusted.

First, we clarify the meaning of the term calibration. According to the discipline of metrology (Bureau International des Poids et Mesures, 2005), calibration is understood as the operation to quantify the deviance of an instrument to a reference. Calibration explicitly does *not* mean adapting the instrument. Calibration is the prerequisite of the adjustment which is understood as the operation of adapting the instrument to reduce the observed deviance. Normally, an adjustment is followed by a re-calibration to check if the adjustment was successful. Transferred to the context of model representativeness, calibration refers to the operation of determining the deviance between the model and the modeled entity itself, whereas adjustment refers to the operation of adapting the model to reduce that deviance. However, in the context of performance prediction (Menascé et al., 1994), model calibration is often understood as the process of adapting the models to improve their accuracy. In other words, in the literature of performance engineering, model calibration and model adjustment are often used synonymously, i.e., are not distinguished.

6.5.1 Model Calibration

In this thesis, calibrating a performance model means comparing the model predictions with measurements of the modeled entity, e.g., a software system. To compare predictions with measurements, the context and conditions of model prediction and system measurement have to be comparable. For instance, a performance model prediction is made for a specific workload and usage scenario, respectively. Thus, to compare the prediction with a measurement, the measurement has to be obtained during a phase where the same specific workload is actually executed. Considering a system at run-time, the challenge of model calibration is to obtain measurements when the system is in a clearly defined state so that the measurements can be used as a reference.

6.5.2 Model Adjustment

In this thesis, adjusting a performance model means improving the model accuracy. Given an observed deviation between measurements and predictions as part of a previous calibration step, in general there are two ways of adjusting a performance model:

- One approach is to increase the model's granularity by changing the model's structure, i.e., refining the model. For instance, one may integrate a representation of a thread pool usage or introduce further parameter dependencies. Adapting specific model entities obviously requires detailed insight about which specific model entities to touch. This knowledge is hard to gain automatically and typically requires manual investigations.

- The other possibility of adjusting a performance model is the adaptation of model parameters. In that case, the granularity of the model does not change. As described in Section 6.4, continuous maintenance of model parameters is an option.

If the deviation determined in the calibration step is systematic, improving the model accuracy is possible without having detailed knowledge about the modeled system. In the following, we describe when and how systematic deviations allow an automatic adjustment of the model parameters (i.e, increasing the resource demands of a performance model) or the predicted metrics (i.e., adding a constant offset to service response time predictions). The result of the adjustment is an increased prediction accuracy.

Algorithm 7 shows when and how resource demands can be automatically adjusted. Let $\overline{Res} = \{Res_0, Res_1, \ldots, Res_n\}$ be the set of resources stressed by a given workload. Let S_{Res_i} be the set of services that are deployed on Res_i, and for each $s \in S_{Res_i}$, let $resourcedemands(s, Res_i)$ be the set of resource demand descriptions of s that stress Res_i. Furthermore, for a resource Res_i, let $U_{Res_i}^{msrd}$ and $U_{Res_i}^{pred}$ be the measured (msrd) and predicted (pred) utilization, respectively. For a service s, let R_s^{msrd} and R_s^{pred} be the measured and predicted average service response time, respectively. The prerequisite for the adjustment is that the predictions of average response time and resource utilization metrics deviate in the same direction, e.g., all the predictions are underestimated. The goal is to adjust the resource demands in a way that the deviations of the resource utilization predictions as well as the deviations of the average response time predictions are reduced. The minimum of the relative deviation of average response time predictions and the relative deviation of utilization predictions is used to obtain a factor that adjusts the resource demands. The factor is calculated for each pair of resource Res_i and service s stressing the resource. The resource demand to be adjusted stems from service s. This algorithm increases model accuracy if the estimated resource demands are biased by a systematic estimation error, e.g., due to the influence of measurement or system overhead.

Another option is to directly adjust the predicted response time metrics, i.e., adapt the results *after* the predictions to add a delay offset. This is useful, e.g., if system requests get delayed until they enter the system boundaries where the measurement sensors are injected. The latter can be the case if the considered component-based software system runs on a complex middleware stack. Let $\overline{S}$ be the set of called services at the system boundary, i.e., the set of services that are called via a SystemCallUserAction as part of a UsageScenario. If $\forall s \in \overline{S} : R_s^{pred} - R_s^{msrd} < 0$ holds, then the delay offset for the response times of $\overline{S}$ is computed as $\min_{s \in \overline{S}}(R_s^{pred} - R_s^{msrd})$. This delay offset is then added to each

```
1  $CalibrateAndAdjustResourceDemands$
2  begin
3      foreach $Res_i \in \overline{Res}$ do
4          if $(U^{pred}_{Res_i} - U^{msrd}_{Res_i} < 0 \land \forall s \in S_{Res_i} : R^{pred}_s - R^{msrd}_s < 0)$
5          $\lor\ (U^{pred}_{Res_i} - U^{msrd}_{Res_i} > 0 \land \forall s \in S_{Res_i} : R^{pred}_s - R^{msrd}_s > 0)$ then
6              $deviation_{Res_i} \leftarrow \frac{|U^{pred}_{Res_i} - U^{msrd}_{Res_i}|}{U^{pred}_{Res_i}}$
7              foreach $s \in S_{Res_i}$ do
8                  $deviation_s \leftarrow \frac{|R^{pred}_s - R^{msrd}_s|}{R^{pred}_s}$
9                  $mindeviation \leftarrow \min(deviation_{Res_i}, deviation_s)$
10                 if $U^{pred}_{Res_i} - U^{msrd}_{Res_i} < 0$ then $factor \leftarrow 1 + mindeviation$
11                 else $factor \leftarrow 1 - mindeviation$
12                 foreach $rd \in resourcedemands(s, Res_i)$ do
13                     $rd \leftarrow rd * factor$
14                 end
15             end
16     end
17 end
```

Algorithm 7: Calibration and Adjustment of Resource Demands

R^{pred}_s with $s \in \overline{S}$. The result is that the deviation of the average service response time predictions from the measured values is reduced.

Note that the described two approaches assume load-independent resource demands. If measurements with different load intensities reveal load-dependent resource demands, then the approaches have to consider the differences between measurements and predictions from low load scenarios up to high load scenarios. If all involved load-dependent resource demands *increase* with the load intensity, considering differences between measurements and predictions from only a low load scenario is sufficient.

6.6 Summary

This chapter presented methods to integrate architecture-level performance models and system environments with the goal to keep them up-to-date during operation as the system evolves. Section 6.1 described monitoring capabilities that are prerequisites to use the presented methods. Section 6.2 described the semi-automatic extraction of architecture-level performance models based on system request tracing, while Section 6.3 described how the model structure can be maintained as part of an autonomic resource management process. Approaches to derive model parameter values are presented in Section 6.4, ranging from the extraction of branching probabilities to the estimation of resource demands, also in virtualized environments. Finally, in Section 6.5, we discussed how architecture-level performance models can be calibrated and adjusted in order to increase their accuracy.

7. Validation

The main goal of the work presented in this thesis is to enable online performance prediction for component-based software systems based on architecture-level performance models. This chapter presents the evaluation of the proposed architecture-level performance modeling abstractions (presented in Chapter 4), the model extraction and maintenance methods (presented in Chapter 6), and the performance prediction mechanisms (presented in Chapter 5).

In Section 7.1, we describe the goals we pursue in the evaluation. The goals are aligned with the success criteria we formulated in Section 1.4.1. Section 7.2 provides an analysis of the trade-off between prediction accuracy and time-to-result of the performance prediction process. Section 7.3 presents a real-world case study with a Software-as-a-Service (SaaS) provider, demonstrating that our approach is applicable to component-based software systems of realistic size and complexity and that the modeling and prediction techniques provide sufficiently accurate performance predictions. Section 7.4 presents evaluation scenarios with the industry-standard SPECjEnterprise2010[1] Enterprise Java benchmark, a representative software system executed in a realistically sized environment. We investigate multiple application scenarios (see Section 1.5) including a scenario where the prediction capabilities are used as a basis to implement an automatic performance-aware resource management approach.

7.1 Evaluation Goals

It is not possible to *prove* that the performance abstractions described in Chapter 4 enable accurate performance predictions for any component-based software system. Recall that a performance model is an abstract representation of the system's performance-relevant properties. The model may neglect important performance properties of the real system, rendering the performance predictions derived from the model inaccurate. Thus, the presented performance abstractions cannot guarantee a certain prediction accuracy. The accuracy varies from scenario to scenario.

[1]SPECjEnterprise2010 is a trademark of the Standard Performance Evaluation Corporation (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 Standard Performance Evaluation Corporation (SPEC) result. The official web site for SPECjEnterprise2010 is located at `http://www.spec.org/jEnterprise2010`.

The goal of the evaluation presented in this chapter is to demonstrate that, aligned with the success criteria identified in Section 1.4.1, (i) the proposed performance abstractions lend themselves to describe architecture-level performance models that are representative in terms of the performance behavior of the modeled systems, (ii) the prediction mechanism of Chapter 5 is capable of deriving performance predictions in online scenarios, and (iii) that the model extraction and maintenance methods described in Chapter 6 are suitable to extract and maintain performance model instances that provide an acceptable accuracy.

In the following, these goals are broken down into several specific evaluation questions to be answered. In Section 7.1.1, we investigate evaluation aspects concerning the modeling capabilities of our approach. Section 7.1.2 describes evaluation aspects concerning the prediction capabilities of our approach.

7.1.1 Modeling Capabilities

An important aspect of the evaluation is whether the proposed architecture-level modeling abstractions are suitable to model the performance-relevant behavior of enterprise software systems in an online context. The question to answer is whether the modeling abstractions have the expressiveness to model the architectural structure and the behavior abstractions? Is it possible to model parameter dependencies as they occur in modern enterprise software systems?

The evaluation of these questions is addressed in two selected case studies. In the first case study we investigate a real-life enterprise software system from a large SaaS provider. Given that the system is of realistic size and complexity, the suitability of our modeling approach can be evaluated in a representative way. The second case study uses the SPECjEnterprise2010 benchmark, a benchmark developed by SPEC's Java subcommittee for measuring the performance and scalability of Java Enterprise Edition (Java EE) based application servers. Given that SPEC is a "non-profit corporation formed to establish, maintain and endorse a standardized set of relevant benchmarks" (SPEC, 2014) with participation of many industrial and academic partners, we consider the benchmark to be representative of an enterprise software system in terms of its performance behavior. In this case study, we evaluate fine-grained modeling abstractions.

7.1.2 Prediction Capabilities

When evaluating prediction techniques, the attained accuracy of the predicted metrics is crucial. In capacity planning, prediction errors of up to 30% concerning the average response time and 5% concerning the resource utilization are considered acceptable (Menascé et al., 1994; Menasce and Virgilio, 2000). In addition to evaluating the prediction accuracy, it is important to investigate if the online performance predictions provide sufficient information to serve as input for autonomic performance-aware resource management. Furthermore, an analysis of the trade-off between prediction accuracy and prediction overhead should be provided.

Another evaluation question is whether the semi-automatic model extraction and parameter estimation techniques described in Chapter 6 provide representative models, i.e., if the predictions based on the extracted models have acceptable accuracy. As further aspects of the prediction capabilities, a sensitivity analysis for parameter variations should be provided as well as an evaluation whether the performance impact of software reconfigurations, deployment changes, or changes in the resource landscape can be predicted.

In the first case study with the real-life SaaS provider, we investigate the prediction accuracy in several evaluation scenarios varying, amongst others, the load intensity, user behavior, and database state. In the second case study with SPECjEnterprise2010, we cover all the mentioned evaluation aspects. We deploy SPECjEnterprise2010 in a representative resource landscape and evaluate multiple variations in our testbed. To assess the prediction accuracy, we compare predicted performance metrics with measured performance metrics, both for response times and resource utilization, in varying scenarios. Variations include the load intensity, the user behavior, the deployment and the resource landscape. Furthermore, in collaboration with Nikolaus Huber, the extracted performance models have additionally been applied in a resource allocation algorithm for dynamic resource allocation in virtualized environments (Huber et al., 2011a; Huber, 2014).

7.2 Trade-Off Between Prediction Accuracy and Prediction Overhead

In this section, we evaluate the tailored performance prediction process (Chapter 5) by analyzing the trade-off between prediction accuracy and time-to-result. Based on a given performance query (Section 5.5), the performance prediction process selects a suitable model abstraction level and model solving technique, and returns the requested performance metrics.

The tailored process is based on three model solving techniques, namely: (i) bounds analysis (Section 5.3.3), (ii) transformation to Layered Queueing Network (LQN) (Section 5.3.2) where the resulting LQN is solved with the analytical solver LQNS (Franks et al., 2011), and (iii) transformation to Queueing Petri Net (QPN) (Section 5.3.1) where the resulting QPN is solved by simulation with SimQPN (Kounev and Buchmann, 2006; Spinner et al., 2012). The latter model solving technique is the only considered solving technique that supports all modeling features we described in Chapter 4. For instance, it is the only solving technique that supports the prediction of percentile response times. To compare the prediction accuracy and prediction overhead of all three solving techniques, we choose an evaluation scenario where all three techniques are applicable. The evaluation has the goal to reveal the individual characteristics of the employed solving techniques, and to analyze if the tailoring mechanism is capable of trading-off between prediction accuracy and prediction overhead. Further evaluation scenarios of that kind are described in Brosig et al. (2014).

Note that in this section the focus in on quantitatively comparing the different model solving techniques, i.e., in this section we do not evaluate the accuracies of the models themselves. For the application of our modeling and prediction approach in real-world case studies, see Section 7.3 and Section 7.4.

7.2.1 Context and Experiment Setup

For the evaluation scenario in this section, we consider a two-tier system as described in the following. An application running on an application server provides a service called *processOrder*. The service is either processed at the application server tier or is delegated to the database. The application server and the database server are each represented by a M/M/1 queue with processor sharing as scheduling discipline.

We provide two service behavior models for *processOrder*, a fine-grained service behavior model and a coarse-grained behavior model. The fine-grained service behavior model is shown in Figure 7.1. There is a branch with two branch transitions, each branch transition is accessed with a probability of 0.5. One branch transition calls database service *processDBS*, the other branch transition contains an internal action that includes a resource demand to the application server.

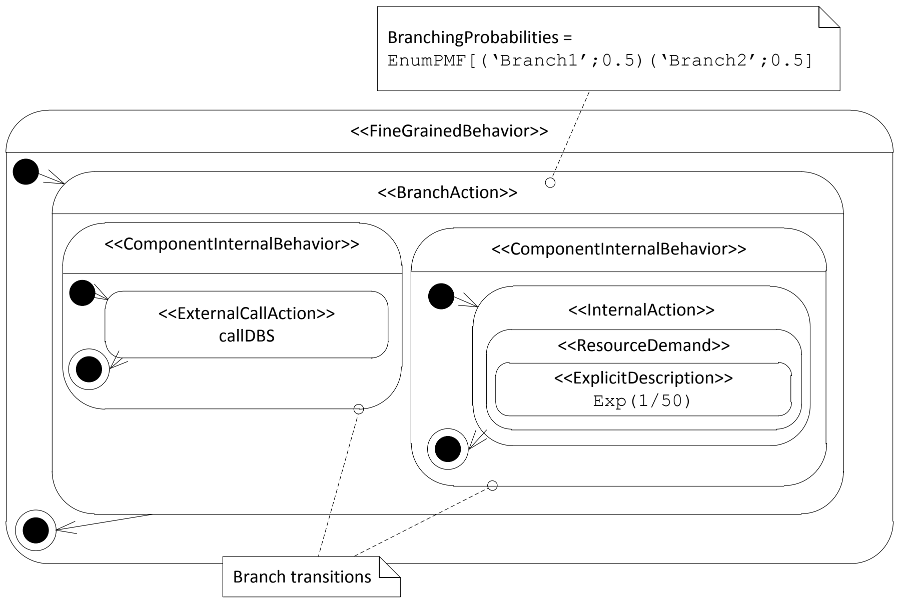


Figure 7.1: Fine-Grained Behavior Model of Service *processOrder*

Figure 7.2 shows the coarse-grained service behavior model of service *processOrder*. It captures the service behavior as observed from the outside of the service providing component. The call frequency of external service *processDBS* is thus described to be 0 with a probability of 0.5, and 1 with a probability of also 0.5. The characterization of the resource consumption of service *processOrder* is also part of the coarse-grained behavior model.

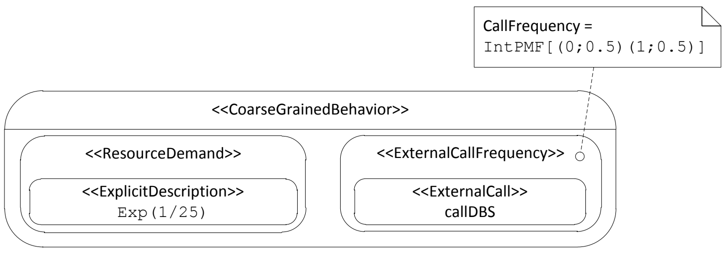


Figure 7.2: Coarse-Grained Behavior Model of Service *processOrder*

In the considered evaluation scenario, the distributions of the resource demands are approximated with exponential distributions. The internal action of the fine-grained behavior is characterized with an exponential distribution with a mean of 50 milliseconds. For the coarse-grained behavior, the mean of the resource consumption is thus 25 milliseconds, and also approximated with an exponential distribution. The service behavior of *processDBS* is modeled as coarse-grained behavior with an exponentially distributed resource demand with a mean of 30 milliseconds. Note that it is common to assume resource demands to be exponentially distributed (e.g., Menascé et al. (1994)). In particular, if resource demands are estimated using techniques that are based on the Service Demand Law (see Section 6.4.3), only mean values are estimated that are typically used to characterize exponential distributions.

We use Descartes Query Language (DQL) to specify a performance query. Listing 7.1 shows a query where the resource utilization of the application server, the resource utilization of the database server, and the average response time of service *processOrder* is requested. In the listing, the query specifies a trade-off between prediction accuracy and prediction overhead using the constraint ‘fast’. In this evaluation, three different trade-offs are investigated. We use the notation ‘fast’, ‘balanced’, ‘accurate’ for the trade-off weights as they are introduced in Section 5.4.2, i.e., the number of different trade-off weights K is set to $K = 3$ where $w_\top = w_3$ maps to ‘fast’, w_2 maps to ‘balanced’ and $w_\perp = w_1$ maps to ‘accurate’.

```
SELECT s.avgResponseTime, app.utilization, dbs.utilization
CONSTRAINED AS ‘fast’
FOR RESOURCE ‘ApplicationServer’ AS app, RESOURCE ‘DBServer’ AS dbs,
    SERVICE ‘processOrder’ AS s
USING connector@location;
```

Listing 7.1: Performance Query “Fast”

We investigate a usage scenario where service *processOrder* is called in an open workload. We consider two different inter-arrival times. The inter-arrival time with a mean of 1/30 seconds (i.e., 30 requests per second on average) corresponds to a high load scenario. The inter-arrival time with a mean of 1/15 seconds corresponds to a low load scenario. For the evaluation, both inter-arrival time distributions are characterized by a normal distribution with a standard deviation of 0.01. For the two load scenarios, we issue performance queries as shown in Listing 7.1, in each load scenario with the three different trade-off specifications ‘fast’, ‘balanced’, and ‘accurate’. Thus, in total, six different performance queries need to be answered.

All experiments executing the analysis and simulation tools were conducted with an Intel Core2Duo CPU at 2.53 GHz. LQNS was available in version 4.3, SimQPN was part of QPME version 2.0.1. For SimQPN, we used the simulation settings as described in Section 5.4.5. For LQNS, the default settings were used: convergence = 0.001, iteration limit = 50, and underrelaxation = 0.5.

7.2.2 Results

The tailored performance prediction process involves a model composition step where the ambiguity of multiple service behavior models for one service is resolved. In the considered scenario, the model composition step has to choose between the coarse-grained behavior and the fine-grained behavior of service *processOrder*. As described in Section 5.4.4, for the two trade-off specifications ‘fast’ and ‘balanced’, the coarse-grained behavior is selected. For the trade-off specification ‘accurate’, the model composition step selects the fine-grained behavior.

Table 7.1 shows the results for the six different performance queries. To compare the different model solving techniques, i.e., bounds analysis (BA), LQNS and SimQPN, the six performance queries are answered by each of the three model solving techniques. Table 7.2 shows the reference results that are obtained using a “gold” SimQPN simulation. For the gold SimQPN simulation, SimQPN’s stopping criteria is set to 1% relative precision, i.e., the simulation is not stopped before enough data is collected to provide a confidence interval for the response time with a half width not exceeding 1% percent of the corresponding mean response time. In the following we first investigate the prediction accuracy, then investigate the prediction overhead, and finally describe the results of the tailoring mechanism.

Prediction Accuracy

As described with the performance query in Listing 7.1, we investigate the average response time of service *processOrder* (R), the utilization of the application server (U_{App}),

Trade-off	Load	Composition	BA			LQNS			SimQPN		
			R[ms]	U_{App}	U_{Dbs}	R[ms]	U_{App}	U_{Dbs}	R[ms]	U_{App}	U_{Dbs}
fast	low	coarseg.	40	0.375	0.225	59	0.375	0.225	43	0.37	0.23
balanced	low	coarseg.	40	0.375	0.225	59	0.375	0.225	44	0.38	0.23
accurate	low	finegr.	40	0.375	0.225	59	0.375	0.225	48	0.38	0.22
fast	high	coarseg.	40	0.75	0.45	118	0.75	0.45	86	0.75	0.45
balanced	high	coarseg.	40	0.75	0.45	118	0.75	0.45	86	0.75	0.46
accurate	high	finegr.	40	0.75	0.45	118	0.75	0.45	101	0.75	0.45

Table 7.1: Prediction Results of Different Model Solving Techniques

Trade-off	Load	Composition	SimQPN (gold)		
			R[ms]	U_{App}	U_{Dbs}
accurate	low	finegr.	49	0.375	0.225
accurate	high	finegr.	96	0.75	0.45

Table 7.2: Reference Results

and the utilization of the database server (U_{Dbs}). The prediction results show the following: Bounds analysis can be used to derive accurate utilization predictions in open workload scenarios. However, it can only provide lower bounds for the average response time since it neglects contention effects in open workload scenarios. The results obtained with bounds analysis are independent of the shape of underlying resource demand or inter-arrival time distributions. LQNS provides accurate utilization predictions but in the considered scenario, it overestimates the response time. This is because LQNS is limited to exponentially distributed resource demands and inter-arrival times. The deviance in the response time prediction stems from the inter-arrival time distribution. LQNS approximates the given normal distribution with an exponential distribution with the same mean. This leads to an overestimation of the contention effects and thus to an overestimation of the response times. Furthermore, we can observe that the average response time prediction provided by LQNS is independent of whether the fine-grained behavior or coarse-grained behavior is selected. SimQPN provides accurate utilization predictions and nearly accurate average response time predictions. The best accuracy is attained where the trade-off specification prefers prediction accuracy to prediction overhead.

Prediction Overhead

The prediction overhead of the individual model solving techniques is obtained by measuring their analysis and simulation times, respectively. Each prediction has been repeated 30 times to obtain an average execution time for each prediction scenario. Bounds analysis provided results within 10ms. The average execution times of LQNS were between 324ms and 339ms in the considered prediction scenarios. The average execution times of SimQPN ranged between 400ms for the 'fast' prediction scenarios and 634ms for the 'accurate' prediction scenarios.

Figure 7.3 shows SimQPN execution times for a prediction scenario where the fine-grained service behavior of service *processOrder* is simulated. For the purpose of a sensitivity analysis, SimQPN is executed with three different simulation stopping criteria settings, corresponding to the trade-off specifications 'fast', 'balanced', and 'accurate' as described in Section 5.4.5. The box plots shown in Figure 7.3 illustrate that in the considered scenario the different stopping criteria settings —with regard to prediction overhead— well reflect the trade-off specifications.

In Brosig et al. (2014), we investigated the prediction overhead also with large performance models in realistic environments. For instance, for the performance model instances as they

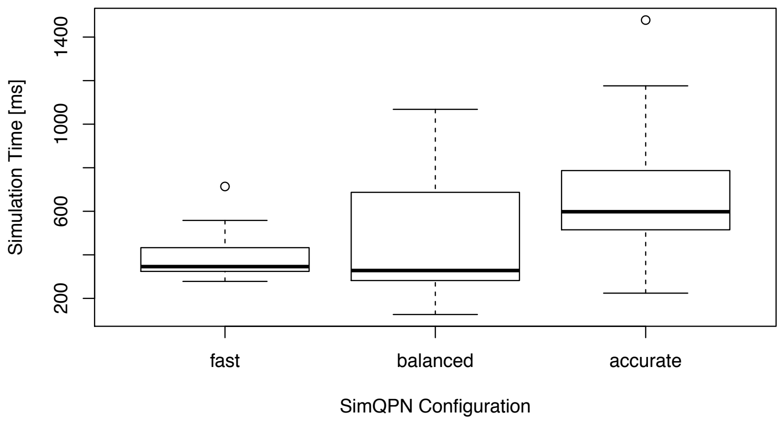


Figure 7.3: Scenario Fine-Grained High Load: Simulation Time with Different SimQPN Settings

are considered in the case study with SPECjEnterprise2010 in Section 7.4, the execution time of LQNS was 0.5 seconds whereas the execution time of SimQPN was 3.8 seconds.

Tailoring

The highlighted cells in Table 7.1 show the results that are provided by the tailored prediction process. For ‘fast’ performance predictions, bounds analysis is chosen as model solving technique. It provides quick results, accurate utilization predictions, but only a lower bound for the average service response time. For ‘accurate’ performance predictions, SimQPN is chosen with a corresponding stopping criteria configuration. The utilization predictions are accurate. The results for the average response time predictions are accurate within a prediction error of 5%. For ‘balanced’ performance predictions, again SimQPN is chosen with a corresponding stopping criteria configuration. Here, the results for the response time are accurate within a prediction error of 10%. Thus, in the considered evaluation scenario, LQNS is not chosen. This is because the inter-arrival time distribution is not exponentially distributed. As shown, an approximation with an exponential distribution as done by LQNS may lead to considerable errors. This is why, in such cases, the tailoring mechanism described in Section 5.4.5 does not consider LQNS as a ‘balanced’ model solving technique. However, in case of exponentially distributed inter-arrival times of open workloads, or in case of closed workloads, the tailoring mechanism considers LQNS as a valid option for ‘balanced’ model solving.

7.2.3 Discussion

We evaluated the tailored performance prediction process (Chapter 5) by analyzing the trade-off between prediction accuracy and time-to-result in an exemplary scenario. The considered evaluation scenario revealed individual characteristics of the solving techniques and showed that the tailoring mechanism is capable of trading-off between prediction accuracy and prediction overhead. In particular, the influence of the shape of the inter-arrival time distribution in open workloads is considered. Further evaluation scenarios of that kind are described in Brosig et al. (2014). The prediction process both supports fine-grained predictions as well as more coarse-grained predictions that come with lower prediction overhead. The tailored performance prediction process encapsulates complex domain knowledge on the employed model solving techniques. The selection of an appropriate model solving mechanism as well as its configuration is done by the performance prediction process.

7.3 Software-as-a-Service Provider Case Study

For a case study in the field, we apply our online performance prediction techniques to a leading SaaS provider. The considered application is part of a component-based large-scale multi-tenant platform, i.e., the platform is shared among multiple customers (tenants). The tenants differ in their size, their number of users, and their application configurations. Providing an enterprise software instance where each day hundreds of thousands of tenants access the same code base and hardware infrastructure requires reliable performance and resource management mechanisms. Performance degradations may have a large impact since they can affect the system's availability for the entire customer base.

In this context, we evaluate the suitability of our modeling abstractions and the capabilities of our performance prediction techniques. The multi-tenant platform and application is described in Section 7.3.1. Section 7.3.2 presents the experiment setup. In Section 7.3.3, we describe the architecture-level performance model we use to derive performance predictions in the scenarios that are explained in Section 7.3.4. Section 7.3.5 summarizes this case study with a discussion.

7.3.1 Customer Relationship Management (CRM)

The considered application is a multi-tenant application sharing one application instance among different customers. Operating only one application instance for multiple tenants promises cost savings, e.g., in terms of hardware resources and maintenance costs (Krebs et al., 2012a). However, the performance management of the single instance requires highest attention, since the single instance represents a single point of failure. Furthermore, the tenants should be able to use the application *isolated* from each other. Each tenant has a negotiated Service Level Agreement (SLA) with the service provider about the performance objectives the services should satisfy provided that the tenant's workload is within an agreed quota. *Performance isolation* means that tenants working within their assigned quota should not suffer from other tenants, even if other tenants exceed their quotas (Krebs et al., 2012b).

The enterprise application we use in our case study is designed to serve hundreds of thousands of customers, possibly at the same time. The application is a Customer Relationship Management (CRM) system, intended to support a tenant's sales department in managing contacts of existing customers as well as prospective new scustomers. This includes managing accounts and contacts of organizations, managing campaigns, managing specific tasks for the sales agents, managing the sales department itself, and so on. The application is customizable, meaning that tenants may configure their own business objects, relations, views and much more. Providing the application in a customizable fashion requires a flexible data model. This imposes further challenges for providing the CRM in a scalable manner with acceptable performance.

Figure 7.4 depicts a simplified architecture of the application. A tenant's user request is first sent to a login server (pool). The login process searches the data center where the tenant's data is located and verifies the user authentication. In case of a successful authentication, the response contains a redirect so that all following requests of the user session are sent to the target data center. Note that the authentication and distribution of tenants to the target data center is not in the scope of this case study since it does not induce a significant performance footprint.

In the data center, arriving requests are load balanced to the responsible tiers. The main tiers are the application tier and the database tier. However, there are several other servers that are responsible for specific tasks such as searching or providing content of large size. Note that the illustration in Figure 7.4 is a simplification and hides much

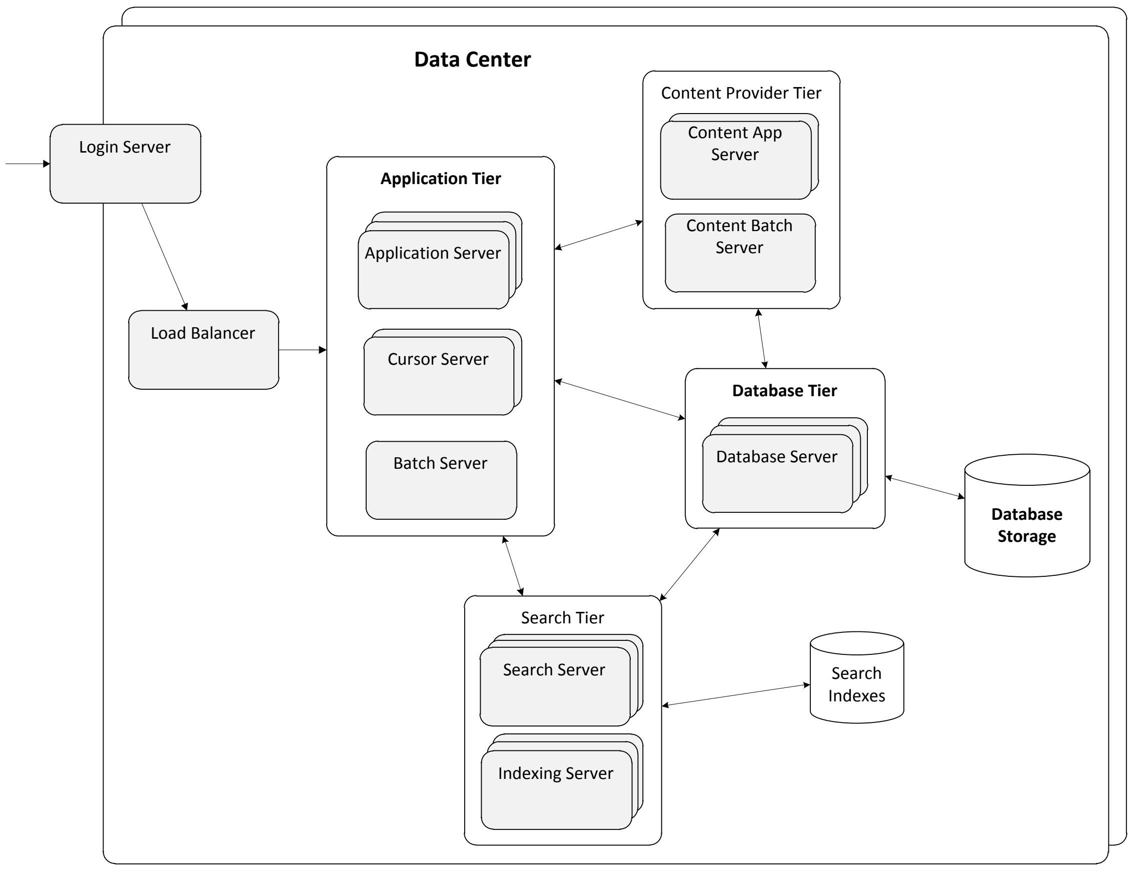


Figure 7.4: Application Architecture

additional complexity, e.g., replica servers. A request is sent to the application tier to be processed by an application server that is part of an application server cluster. It is important to note that the application servers are *stateless*, i.e., adding or removing application servers while the system is running is possible without any disruption. Thus, scaling the application server cluster is easy. The request then typically translates into database transactions. Since most of the requests involve database access, the database tier is the most critical resource. The database tier is realized by a database cluster. However, given that the database is *stateful*, the cluster is not used for a dynamic load balancing among the database servers. Instead, the database cluster is used to improve the system reliability. In case of a failure of a database server instance, other servers of the cluster can take over the load. However, such a failover comes with additional overhead since the database cluster needs to ensure data consistency. During normal operation, the so-called interconnect traffic between the database nodes is reduced to a minimum. Typically, database requests belonging to one tenant are processed by the same database server instance in the cluster. This makes an efficient data organization (e.g., efficient caching) of tenant-specific data within the database instance possible. For the same reason, the data stored in a Storage Area Network (SAN) is partitioned by tenant id.

However, given the high database load, there are many efforts to reduce the load on the database. For instance, searching is performed by a separate search tier. The search tier consists of indexing servers that create search indexes in the background and search servers that actually process search queries. Furthermore, business data objects of large size are not handled by the database servers but by separate content providers. Moreover, a request's database cursors (a database cursor represents a query result) are not stored in the database, but on separate cursor servers. In general, processing tasks are attempted to be split up in parts that need to be processed in a synchronous fashion and parts that can be processed asynchronously. Asynchronous or background tasks are then processed by additional batch servers.

To sum up the architectural overview, the SaaS provider tries to distribute tasks of the database layer on several machines where possible. The data is partitioned by tenant id in order to enable efficient caching mechanisms. Furthermore, if possible, tasks are processed preferably as *bulk* operations instead of single individual operations.

In the following, we describe an additional performance optimization that is relevant in the considered evaluation scenarios. As explained, database performance is critical. Today's database servers typically apply query optimizations when a database query is issued. The goal is to find a query execution plan that promises fast query processing. The optimization is based on relevant statistics about the target table(s) and index data. However, given that conventional databases are designed for single-tenant applications and do not consider multiple tenants, the statistics delivered by the database servers are aggregated over multiple tenants. Thus, a query optimization fails to account for the tenant's data characteristics, i.e., the optimization is not tailored to the tenant who issues the query. Given that the tenants may largely differ in their sizes, considering a tenant's data characteristics yields promising performance optimizations.

To provide tenant-aware optimization statistics, the SaaS provider maintains its own set of optimizer statistics. Optimizer statistics are kept for each data object. Gathered statistics involve not only the tenant level but also the user group level as well as the user level. This makes it possible to, e.g., consider the amount of data rows that the query issuing user is allowed to access. Based on this information, the user query is preprocessed. The output of the preprocessing is a database query that is tailored, amongst others, to the corresponding tenant, possible filter criteria, and user group or user access rights. The same query can thus be processed in different ways. The query's filter selectivity and the user's access rights may in one case suggest to use an ordered hash join to access the data or in another case suggest to use a filter-specific index. The preprocessing step is prepended to the conventional query processing of the database. Obviously, the preprocessing incurs additional overhead, but the performance gain predominates. Since the query processing depends on parameters such as tenant id, user group and user access rights, the same query can result in different performance behavior. This behavior should be considered when building a performance model.

7.3.2 Context and Experiment Setup

For SaaS providers, a reliable performance and resource management requires the ability to answer questions concerning capacity planning, admission control, SLA management, and energy management. Performance predictions help to answer questions such as:

- How to adapt the size of the application server cluster to the weekly workload cycle? How many application servers are needed for a certain workload intensity?
- Which data center and which database server instance should a new tenant be assigned to?
- Can a given SLA for a certain type of request be guaranteed in terms of response time objectives?
- Can a tenant increase its request rate without affecting the performance of other tenants? Will there be an overload situation if the request arrival rate of the tenant increases?

- How will the application performance develop as the data volume of a tenant increases over time?

In the following case study, we evaluate scenarios of the core CRM application. To stress the database, we use scenarios representative of a *large-scale* tenant. Large-scale means that the tenant has many users and large amounts of data, e.g., the largest business object allocates approximately 500 Gigabytes in the database. We evaluate both read and write workloads as well as bulk operations. To stress the application server, we use a benchmark specific for the application tier. In addition, we evaluate several mixes of the mentioned database intensive and application server intensive workloads. The search tier or the content providing servers are not stressed. As part of the evaluation, we build an architecture-level performance model, conduct predictions varying the workload intensity and service input parameters, and compare the predictions with measurements on the real system. In this way, we evaluate if the proposed modeling abstractions are suitable to obtain accurate performance prediction in a representative online context.

To not disturb the productive environment, we cannot run experiments in the production environment. Thus, experimentation requires a *production-like* experimental environment. We consider an experimental environment to be production-like in terms of its performance behavior, if the following holds:

- The hardware infrastructure of the experimental environment resembles the infrastructure that is used in production.
- The deployment of the application and the used software stack in the experimental environment is the same as in the production environment.
- The state of the experimental environment, i.e., the database state, resembles a production database state.
- The workload that stresses the system resembles production workload.

For performance testing purposes, the SaaS provider has several such production-like environments available. Furthermore, we use a performance testing framework that orchestrates measurement experiments. The framework brings the experimental environment in a defined state before performance measurements are conducted (ramp-up phase), it configures and triggers the workload driver JMeter[2], it configures and triggers the logging infrastructure, and it is responsible for the ramp-down of the experiment. Moreover, it collects and aggregates measurement results and log data from the involved server instances. The logging infrastructure uses Splunk[3]. It is important to note that this is the same logging system that runs in production. For our evaluation scenarios, it is important to note that we do not introduce new log monitors or new instrumentation points, we work with a log configuration that resembles the log configuration of the production environment. It is an explicit goal of this case study to work with the log configuration of the production environment to evaluate how the existing monitoring data can be fit to the provided modeling abstractions.

As mentioned, the database state needs to be in a representative state. However, due to confidentiality policies it is not possible to use actual production data. We use synthetic, anonymized production data instead. The anonymized data is representative with respect to the number and kind of tenants that are served by the database server. Tenant data differs in the type and amount of data. For instance, it is important to not only provide realistic numbers of business objects, but also realistic relations between the business objects. For the purpose of performance testing, the SaaS provider has a representative

[2]http://jmeter.apache.org/
[3]http://www.splunk.com/

synthetic dataset available. The representativeness is verified on a regular basis using data characteristics obtained with the logging infrastructure.

The resource environment is depicted in the following before we describe the architecture-level performance model instance and how we built it. Afterwards, we compare performance predictions provided by the model with measurements in the experimental environment considering different scenarios.

Resource Environment

For the evaluation, we conducted experiments in a production-like resource environment as depicted in Figure 7.5. Note that the workloads that have been provided by the SaaS provider do not stress the content provider tier or the search tier and thus, those tiers can be neglected. Nevertheless, the two application server instances as well as the database instance and the SAN were of the same type and configuration as the production system. The database server has 48 CPU cores and runs an Oracle Database, the application server instances each have 24 CPU cores where Jetty[4] is running as application server. The load balancer is an F5 load balancer[5], the switch is from Juniper Networks[6]. Basically, the experimental environment can be understood as a *vertical slice* of the production environment, i.e., instead of ≈ 20 application servers the experimental environment provides 2 application servers, and instead of eight database servers the experimental environment provides one database server. Given that, as described in Section 7.3.1, the application tier is *stateless* and the database tier is configured in a way that a tenant is served always by the same database node, the experiment setup can provide measurement results that are representative for production environments.

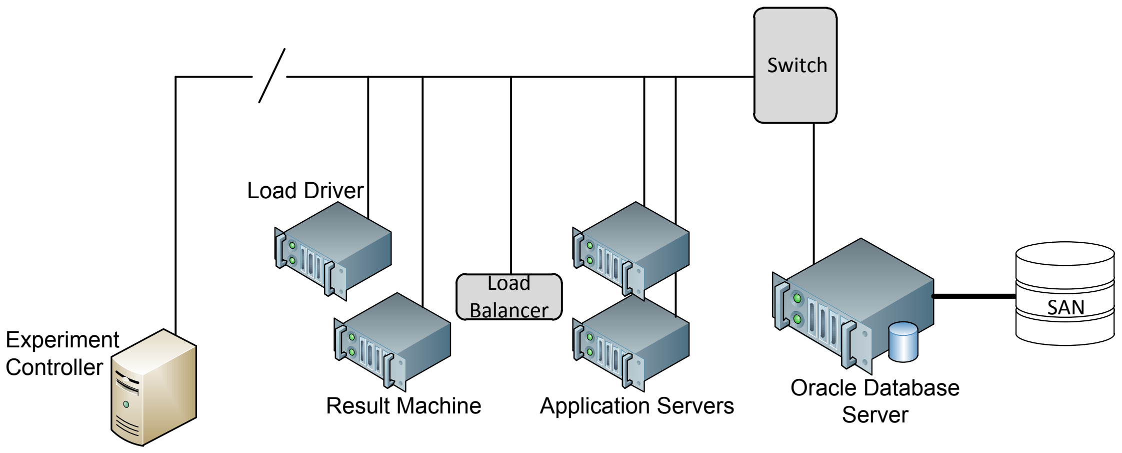


Figure 7.5: Resource Environment

The load driver machine running JMeter as well as the result machine are of the same hardware characteristics as the application server instances. The experiment controller that coordinates the measurement experiments is a workstation having a twelve core CPU. The CRM system is deployed on the application servers and the database server as in production. The database state resembles the production database state. However, the data itself is anonymized due to confidentiality policies.

[4]http://www.eclipse.org/jetty/
[5]https://f5.com/
[6]http://www.juniper.net

7.3.3 Architecture-Level Performance Model

Two business scenarios derived from real-world problems of the SaaS provider are considered: On the one hand, we model the task management business scenario that involves activity lists for sales agents where the sales agents are hierarchically structured in groups. On the other hand, we model an integration service, a service to create and update business objects that are concerned with the end-customers of a tenant. This includes the business objects: household, account (representing an end-customer), as well as relationships between accounts (i.e., if and how the customers are related with each other), sales agent roles, and sales agent activities. An integration service is accessible via Simple Object Access Protocol (SOAP), and allows to process several records per integration request. Thus, the integration service implementation makes use of database bulk operations. Furthermore, we consider a benchmark that stresses specifically the application tier by generating complex UI forms.

Figure 7.6 depicts a high-level overview of the structure of the architecture-level performance model. The system model shows a load balancer that distributes incoming requests to one of the two CRM instances that themselves need a database instance. A CRM instance refers to a composite component that in turn consists of component instances of, e.g., the component providing the integration service. Note that the *Entry* component represents the entry point for the modeled services of the CRM system. In contrast to the SPECjEnterprise2010 case study presented in Section 7.4.2, in this case study we model the component services with coarse-grained behavior descriptions. Given that the CRM services highly depend on meta-data that describes the tenants' customizations, a fine-grained service control flow is hard to model.

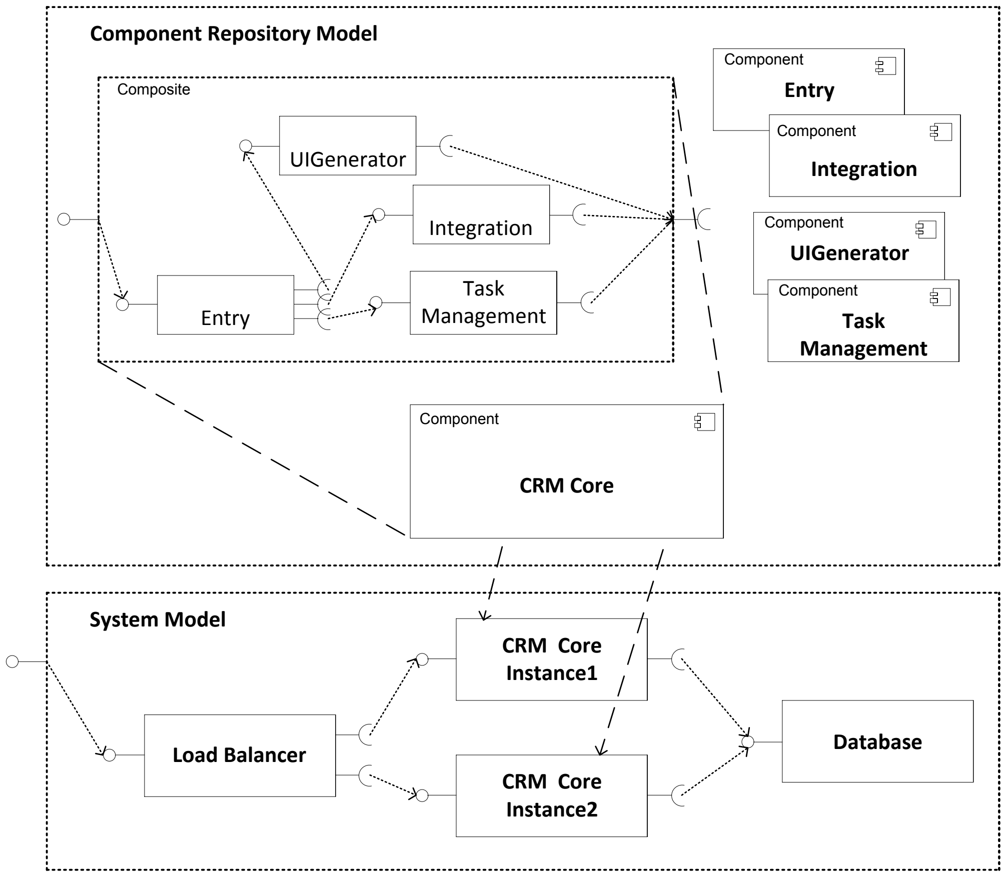


Figure 7.6: Application Architecture Model

For the model parameter estimation, we use monitoring statistics as they are captured in the production system. Thus, we do not inject additional monitoring overhead compared to the production system. The resource demands were estimated during performance tests executed with a steady state time of 900 seconds and a warm-up time of 600 seconds. CPU resource demands are approximated using response time measurements (see Section 6.4.3) that are obtained in a low load scenario, i.e., in a scenario where both the application server CPU and the database server CPU load is below 20%. For each request, we measure the CPU time on the application servers, the CPU time on the database server, and the I/O delay as observed at the database.

7.3.4 Results

The target of this case study is to evaluate if the proposed modeling abstractions are suitable to obtain accurate performance prediction in a representative online context. As mentioned in Section 7.3.3, the investigated evaluation scenarios capture various entities of the CRM core application. To stress the database, we use scenarios representative of a large-scale tenant with large amounts of data. We consider both read and write workloads as well as bulk operations. To particularly stress the application server, we use a specific benchmark for the application tier.

In the scenarios of this case study, we vary the workload intensity, workload mixes, and service input parameters. Deployment changes are not considered, as the SaaS provider does not (yet) apply repetitive deployment changes in the context of dynamic performance management. The performance metrics of interest are average response times, response time percentiles, and average resource utilization of the application server CPUs (APP CPU) and the database server CPU (DBS CPU). In each scenario, we parameterize as well as calibrate and adjust the architecture-level performance model under low load conditions ($\approx$ 20% CPU utilization), and then conduct predictions for medium load conditions ($\approx$ 40%), high load conditions ($\approx$ 60%), and very high load conditions ($\approx$ 80%), comparing the results with steady-state measurements on the real system. The measurements are obtained during a steady-state time of 900 seconds, with a warm-up phase of 600 seconds and a ramp-down phase of 300 seconds.

Scenario 1: Task management. As first scenario, we consider task management services. The usage profile is shown in Figure 7.7. In an open workload, a sales agent logs-in and browses through several activity lists. There are activity lists directly assigned to the sales agent (*MyActivities*), lists assigned to the sales agent's team (*MyTeamsOpenActivities* and *MyTeamsClosedActivities*), and a list of all activities (*AllActivities*). Note that all the list views depend on the specific sales agent and his/her position in the hierarchy, i.e., the list of all activities may differ from sales agent to sales agent due to different visibility settings. The JMeter load driver script of the task management workload ensures that the usage profile is executed for different sales agents. Given that the login and logout requests are insignificant compared to the list views, in the following, we focus on the response time metrics of the activity list views.

Figure 7.8 a) shows the measured and predicted server CPU utilization for the different load levels. The utilization of the DB server varies from 20% to 80%, the application tier is only little utilized. The utilization predictions fit the measurements very well. Figure 7.8 b) shows the average response times of the four activity list views for the four load levels. The average response times vary between 200 and 2000 milliseconds, with list view *AllActivities* having the slowest response time, and *MyActivities* the fastest response time. As expected, the higher the load, the faster the response times increase. The lines in the figure illustrate the gradients between the four load stages. Note that the y-axis is of linear scale, but there is a gap between 700 and 1700 milliseconds. In Figure 7.8 c),

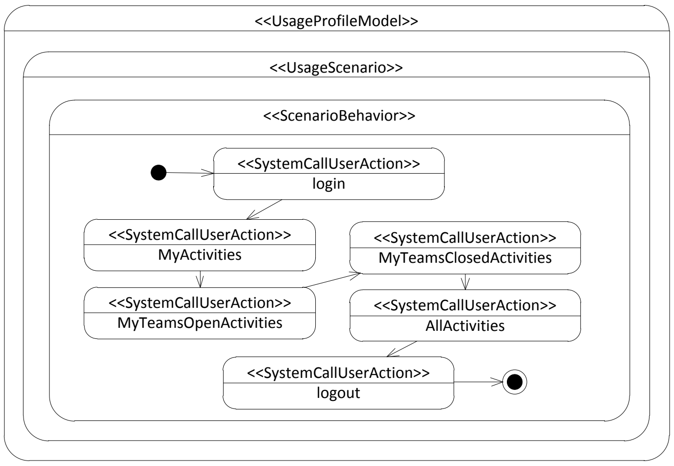


Figure 7.7: Scenario 1: Task Management Usage Profile Model

the relative error of the average response time predictions is shown. It shows that the relative error is below 20% across all load levels. The error is computed relative to the measurements, e.g., a measurement of 100ms and a prediction of 90ms would result in a relative error of 10%.

In the low load scenario, the prediction error is zero since the model is calibrated and adjusted with the measurements of this scenario. The calibration and adjustment for each service works as follows: As offset, we compute the difference between measured average service response time and predicted average service response time in the low load scenario. This offset represents the connection overhead introduced by, e.g., network delay, and is then added as constant offset to the predictions for each load level.

Figure 7.9 a) shows the 90th percentile response time for the different load levels. The trend of the percentiles from low load to very high load is similar to the trend of the average response times shown in Figure 7.8 b). As expected, the 90th percentiles are higher than the average response times. In case of service *MyActivities*, the difference between the average response time and the 90th percentiles is small. Apparently, the average service response time of *MyActivities* is biased by few longer-lasting requests. However, the predictions of the percentiles fit the measurements with an error of mostly below 20% as depicted in Figure 7.9 b). The largest prediction error is observed for service *MyActivities*.

Figure 7.10 a) shows the empirical distribution of the response time of service *AllActivities*, measured at the high load level. The histogram clearly exhibits multi-modality, indicating the complexity of the *AllActivities* service. The multi-modality stems from the fact that different sales agents have different permissions with regard to the question which activities they are allowed and which activities they are not allowed to see. The performance model does not explicitly model the different permissions because of their underlying complexity, but given that the resource demands are estimated using response time approximation, the predicted distribution of service *AllActivities*, depicted in the histogram in Figure 7.10 b), reflects the measured behavior quite well.

Scenario 2: Integration services. In the next scenario, we consider the integration services which are needed in order to keep business objects of the CRM system up-to-date with an external Enterprise Resource Planning (ERP) system. The considered integration services capture the business objects *Household*, *HouseholdMember*, *Account*, *AgentRole* and *Relationship*. Each of the business objects has an external id, representing the object's identity in the external ERP system. The ERP system acts as source, the CRM is the

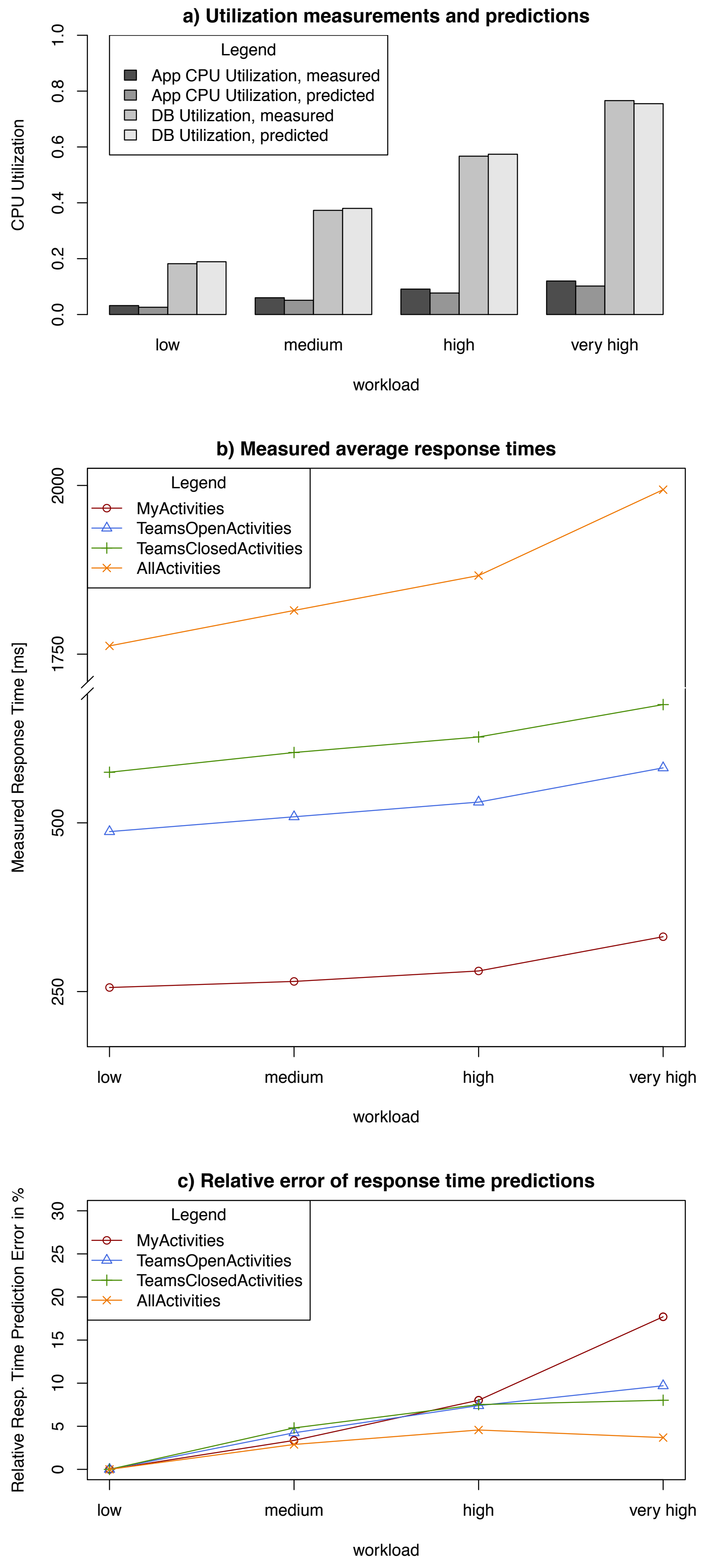
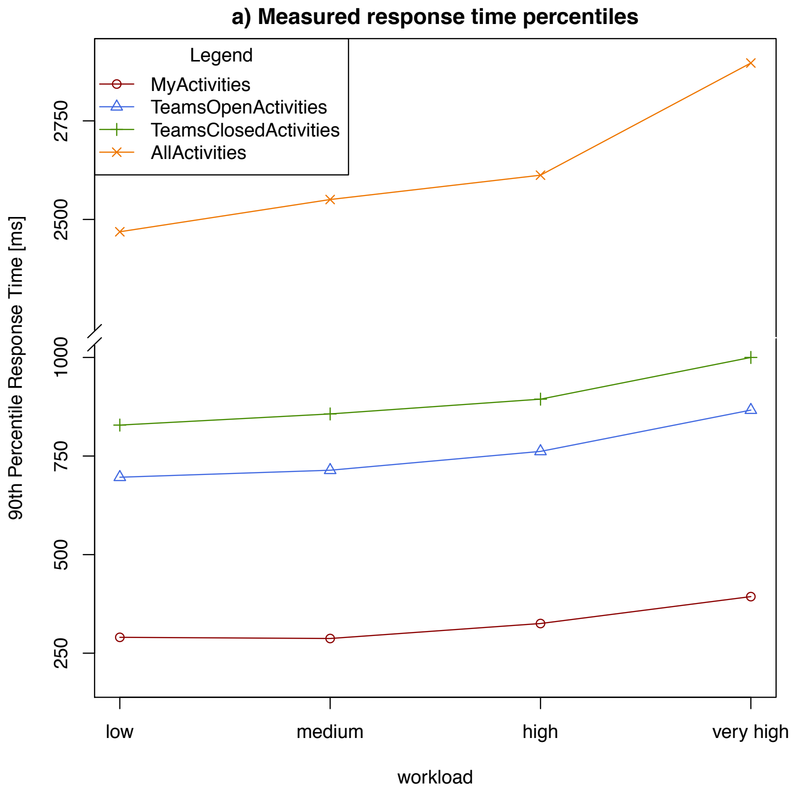


Figure 7.8: Scenario 1: Measurements, Prediction Results and Prediction Errors



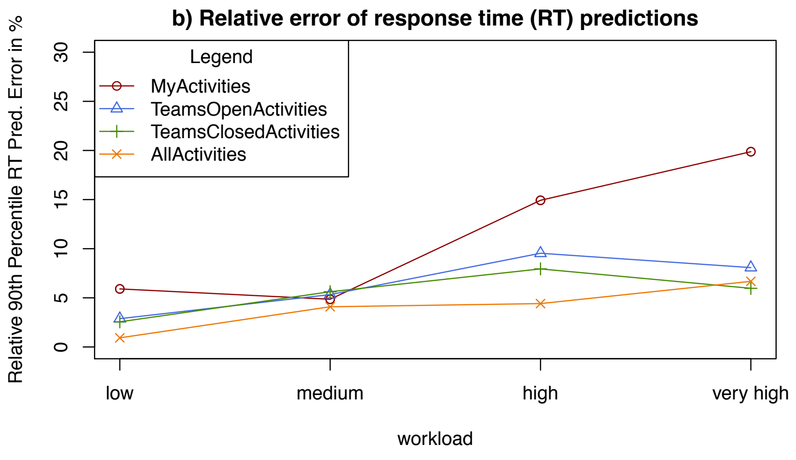
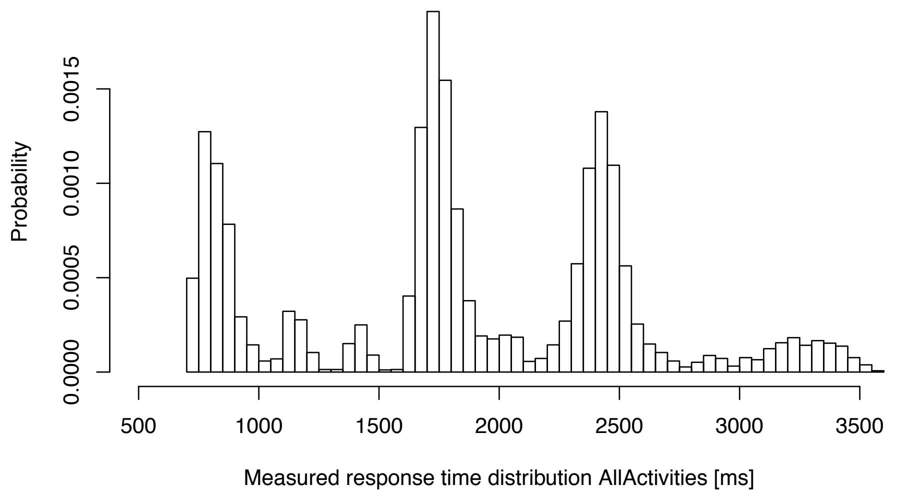


Figure 7.9: Scenario 1: Measurements and Prediction Errors of 90th Percentile Response Times



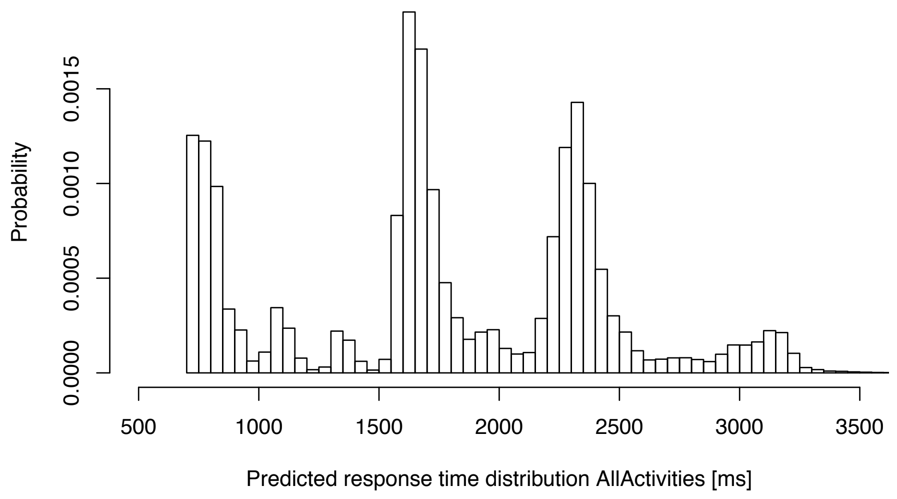


Figure 7.10: Scenario 1: Measurement and Prediction of *AllActivities* Response Time Distribution for the High Load Level

destination. The usage profile is shown in Figure 7.11. For each business object there is a separate usage scenario where the corresponding integration service is called in a loop. The payload of the integration services is not modeled, it consists of a list of 200 records describing the business objects to update. The usage profile serves as a load test for the integration services. The JMeter load driver script ensures that the records to be updated are different for each integration service call. Thus, in contrast to the workload considered in Scenario 1, Scenario 2 is a scenario that stresses mostly database writes.

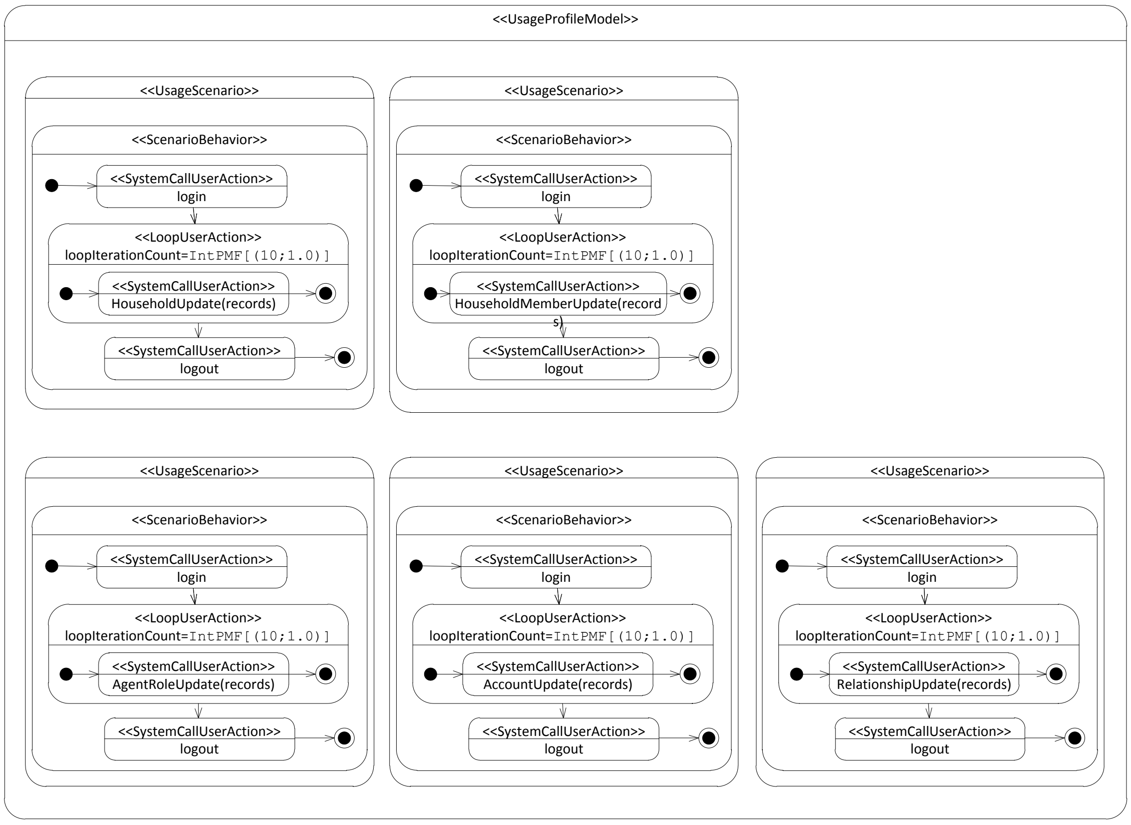
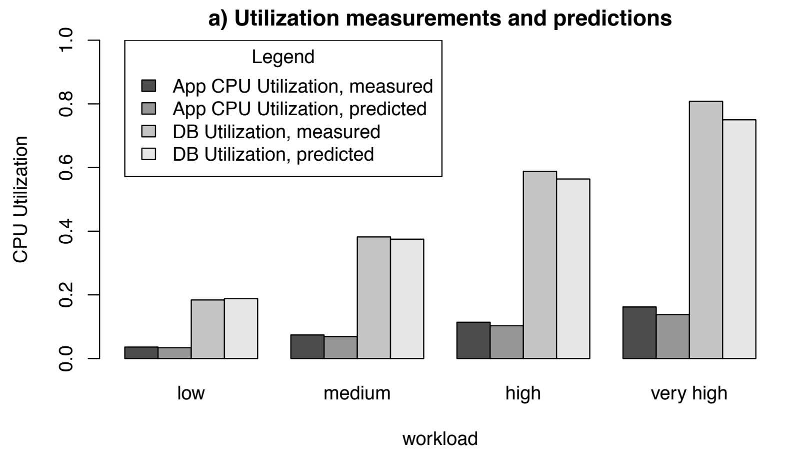
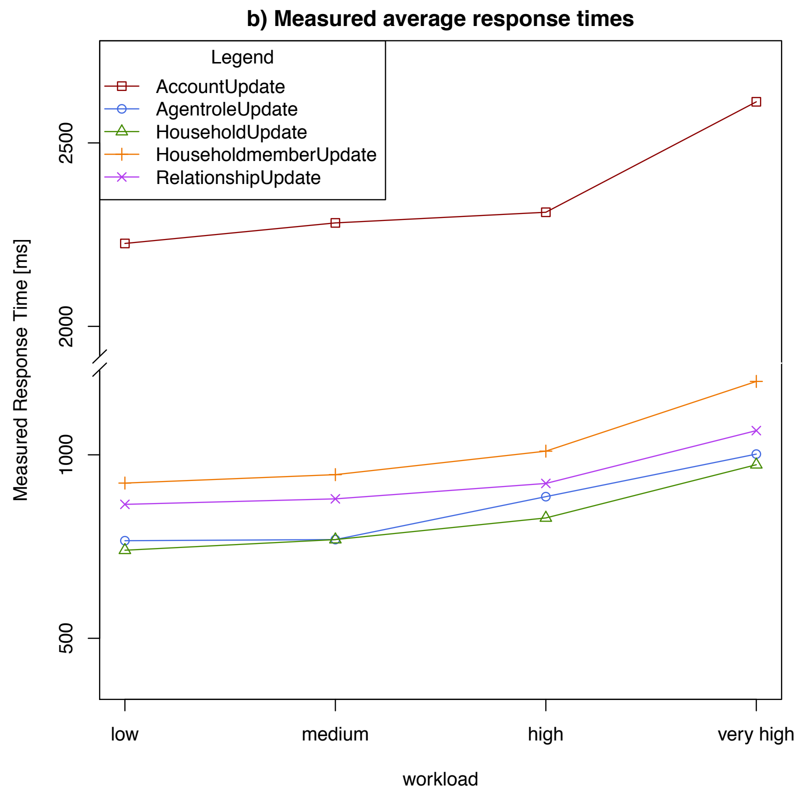

<<UsageProfileModel>>
<<UsageScenario>>
<<ScenarioBehavior>>
<<SystemCallUserAction>> login
<<LoopUserAction>> loopIterationCount=IntPMF[(10;1.0)]
<<SystemCallUserAction>> HouseholdUpdate(records)
<<SystemCallUserAction>> logout
<<SystemCallUserAction>> HouseholdMemberUpdate(records)
<<SystemCallUserAction>> AgentRoleUpdate(records)
<<SystemCallUserAction>> AccountUpdate(records)
<<SystemCallUserAction>> RelationshipUpdate(records)

Figure 7.11: Scenario 2: Integration Usage Profile Model

Given that the login and logout requests are insignificant compared to the list views, in the following, we focus on the response time metrics of the five integration services. Figure 7.12 a) shows the measured and predicted server CPU utilization for the different load levels. The utilization of the DB server varies from 20% to 80%, the application tier is only little utilized. The utilization predictions fit the measurements quite well. For the highest load level, the predictions underestimate the measured utilization by 7%. Figure 7.12 b) shows the average response times of the five integration services for the four load levels. The average response times vary between 700 and 2600 milliseconds, with *AccountUpdate* having the slowest response time, and *HouseholdUpdate* the fastest response time. As expected, the higher the load, the faster the response times increase. The lines in the figure illustrate the gradients between the four load stages. Note that the y-axis is of linear scale, but there is a gap between 1250 and 1900 milliseconds. In Figure 7.12 c), the relative error of the average response time predictions is shown. The relative error is below 20% across all load levels. As in Scenario 1, the error is computed relative to the measurements, and in the low load scenario the prediction error is zero since the model is calibrated and adjusted with the low load measurements.

In the integration workload above, the number of records per request is constantly set to 200. Now we vary the number of records per request and investigate service response





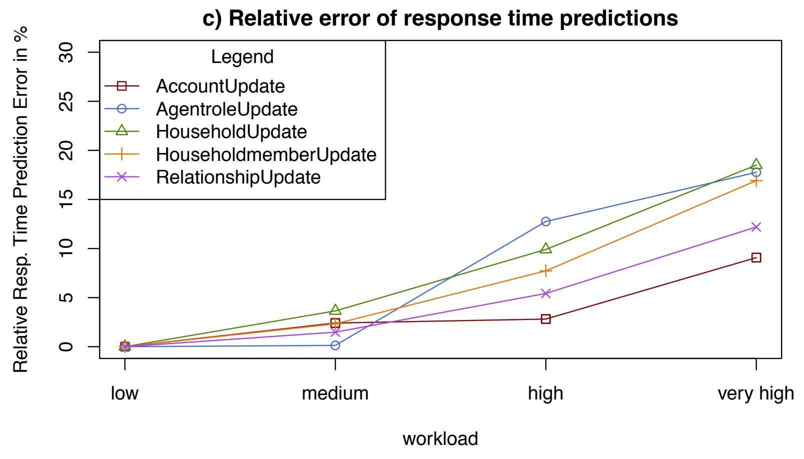


Figure 7.12: Scenario 2: Measurements, Prediction Results and Prediction Errors

times and resource utilization for these variations. For each integration service, there are two parameter dependencies. There are dependencies between the service input parameter *records* (more specifically, the number of elements of parameter *records*) and the service's resource demands at the application server and at the database server. This dependency cannot be explicitly characterized, but empirically characterized using monitoring statistics. The monitoring statistics that are used for characterizing the parameter dependency are captured in workload runs under low load where the number of records per request varies randomly between 25 and 200.

Table 7.3 shows two different variations of the number of records. In variation A, 40% of the integration service requests have 25 records, 30% of the requests have 50 records, 20% of the requests have 100 records, and 10% have 200 records. Thus, requests with small record counts dominate in variation A. In variation B, half of all integration service requests have a record count of 200, 30% have a record count of 100, and 20% have a record count of 50. The workload intensity in terms of request arrival rate is the same for both variations.

	records per request			
variation	25	50	100	200
A	40%	30%	20%	10%
B	0%	20%	30%	50%

Table 7.3: Scenario 2: Parameter Variations

Figure 7.13 a) shows the measured and predicted server CPU utilization for the two variations. For both variations, the database resource is the dominant resource. As expected, the utilization for variation B is higher than for variation A. The utilization predictions fit the measurements within an error of 10 percent. This time, the predicted utilization overestimates the measured utilization. Figure 7.13 b) shows the 90th percentile response times of the five integration services for the two variations. The percentiles vary between 300 and 2300 milliseconds. Comparing the percentiles for the variations A and B, the percentiles for variation B are more than two times the percentiles for variation A. Note that the y-axis has a gap between 1200 and 1900 milliseconds. In Figure 7.12 c), the relative error of the 90th percentile response time predictions is shown. The relative error is mostly below 20%. Only service *AccountUpdate* has a prediction error of about 25%. As in the other scenarios, the error is computed relative to the measurements.

Scenario 3: Application tier benchmark. In the next scenario we consider the application tier benchmark, in the following also denoted as AppBench. AppBench is designed to stress specifically the application tier by generating complex UI forms. The usage profile is shown in Figure 7.14. Between a login and logout, two kinds of requests are repeatedly generated, namely *Generate_Get* and *Generate_Post*. The JMeter load driver script randomly issues various form requests in order to stress the application server stack in different ways.

Given that the login and logout requests are insignificant compared to the generated *Generate_Get* and *Generate_Post* requests, in the following, we focus on the response time metrics of the two AppBench requests. Figure 7.15 a) shows the measured and predicted server CPU utilization for the different load levels. The utilization of the application servers varies from 10% to 80%, the database server is only little utilized. The utilization predictions fit the measurements only for low and medium load. In the high and very high load levels, the predicted application server utilization clearly underestimates the measured utilization. This indicates that there are load-dependent resource demands that are not accurately represented in the performance model. Figure 7.15 b) shows the average

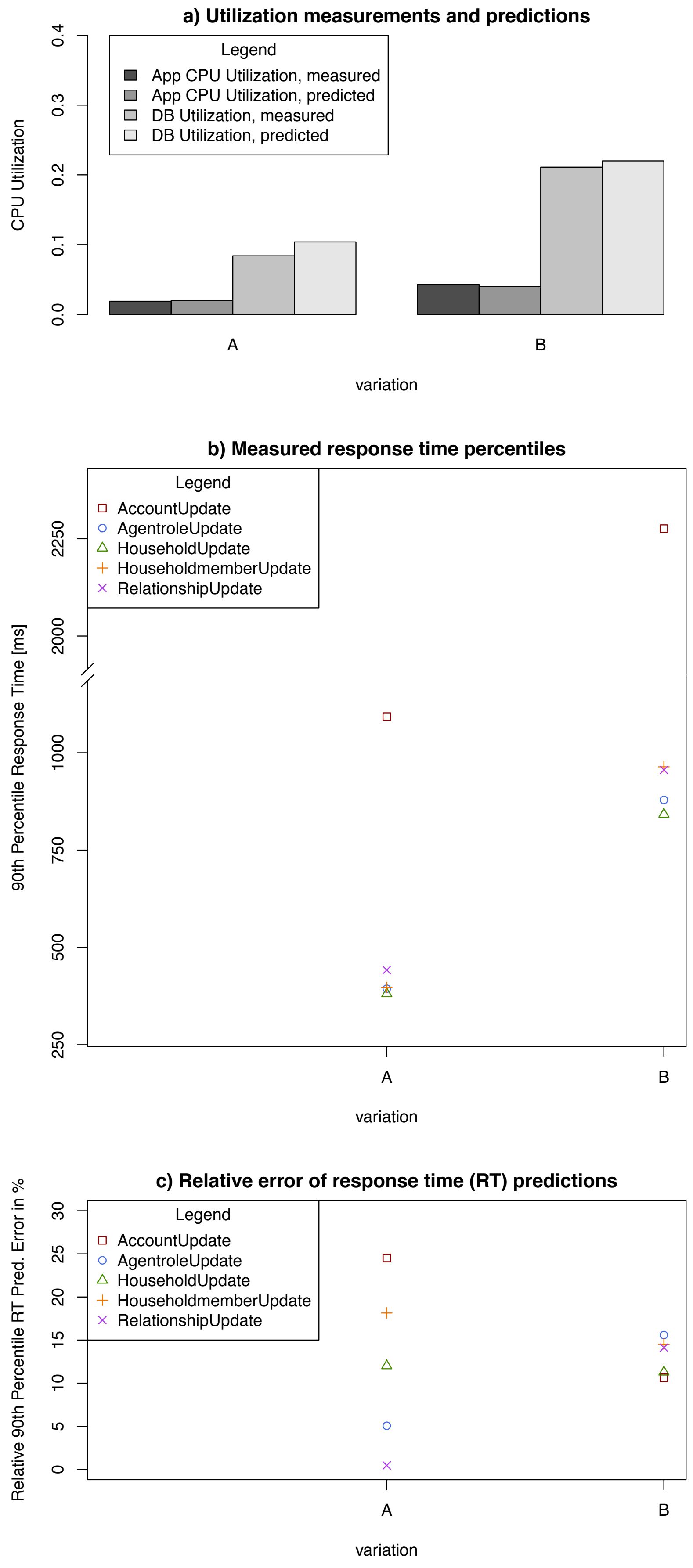


Figure 7.13: Scenario 2: Measurements, Prediction Results and Prediction Errors

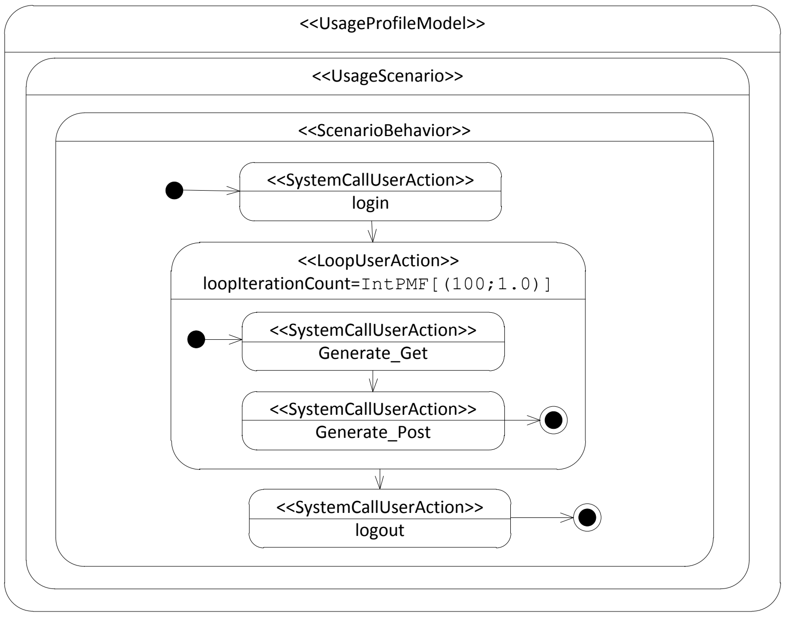


Figure 7.14: Scenario 3: Application Tier Benchmark Usage Profile Model

response times of the two considered requests for the four load levels. The average response times vary between 40 and 170 milliseconds, with *Generate_Get* having the slowest response time, and *Generate_Post* the fastest response time. The lines in the figure illustrate the gradients between the four load stages. While in the low load and medium load levels, the response times increase only slowly, they are almost doubled in the high load and very high load stages. In Figure 7.15 c), the relative error of the average response time predictions is shown. For the high load and very high load levels, the relative error is up to 50%. The high response time increase under high load is thus not predicted properly.

The reason for the high deviation of predictions and measurements is the load-dependent performance behavior. An investigation of the application server monitoring statistics reveals that the Java garbage collector (Hunt and John, 2011) runs many full garbage collections in high load conditions. These full garbage collections, on the one hand, explain the additional load on the application servers, on the other hand, they explain the delayed response times. The garbage collector is not reflected in the performance model, and is thus not considered in the performance prediction. We intentionally refrained from providing a garbage collector model for the following two reasons. On the one hand, a garbage collector model would be specific for a certain garbage collection algorithm with a certain set of configuration parameters, and it can thus easily become outdated with, e.g., an upgrade of the Java Virtual Machine (JVM). On the other hand, even if a garbage collector model was available, in practice it is often infeasible to obtain the relevant input parameters. A garbage collector model requires information about the state of the Java heap as well as the requests' presumed memory consumption. However, this detailed information is typically not available at run-time. Nevertheless, situations where garbage collection leads to significant performance degradations are to be avoided. Using our utilization predictions for the application server together with a specified threshold for the application server utilization (when garbage collection typically leads to performance problems), such situations can still be anticipated.

Scenario 4: Workload mixes. In this scenario, we consider several workload mixes of the task management, integration services and AppBench workloads investigated above. This way, we combine database-intensive read and write workloads with an application server-specific workload. Table 7.4 shows five examined workload mixes. The workload

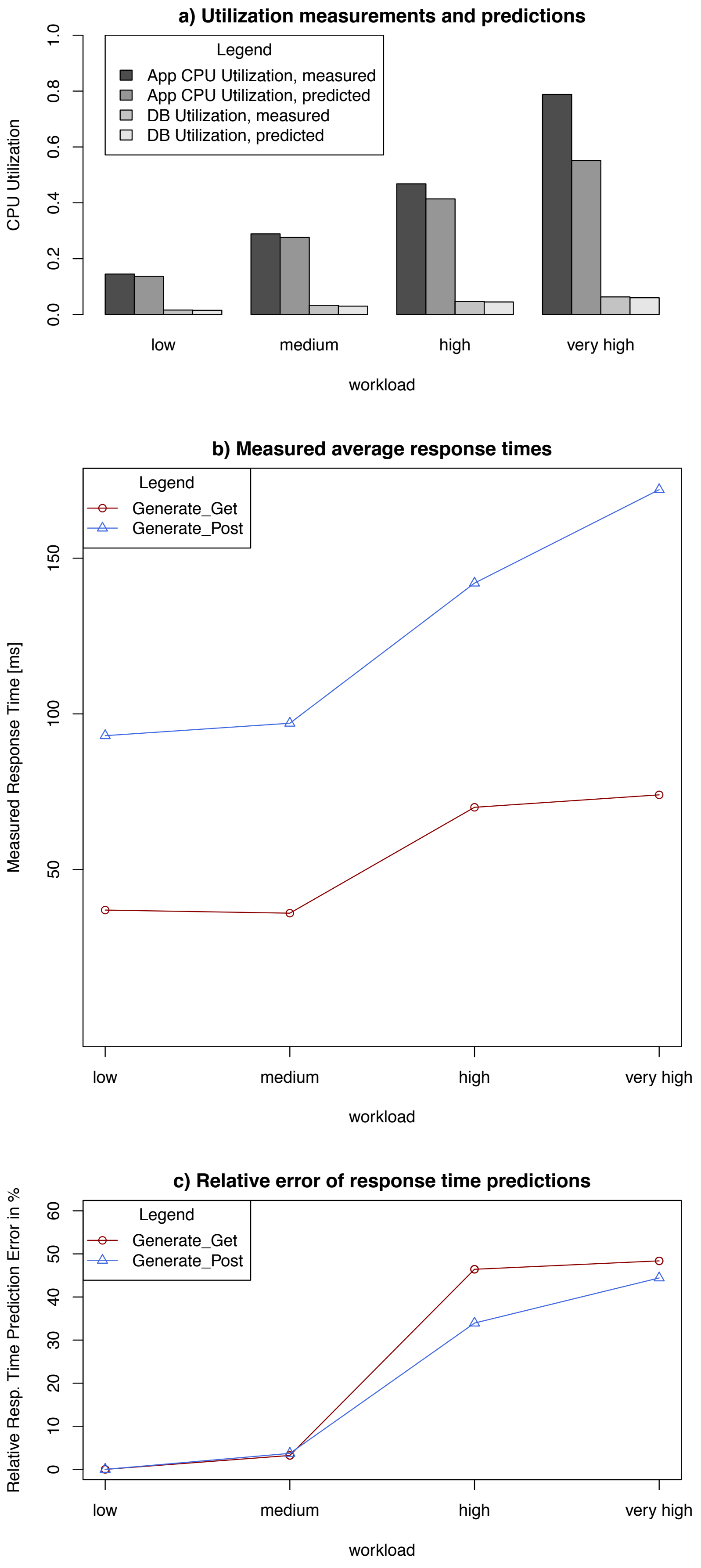


Figure 7.15: Scenario 3: Measurements, Prediction Results and Prediction Errors

intensity levels low, medium, high and very high correspond to the intensity levels analyzed as part of Scenario 1, Scenario 2 and Scenario 3. For instance, workload mix 1 is a combination of the three workloads at low load level. Workload mix 4 is a combination of a low load task management workload, a high load integration workload, and a high load AppBench workload.

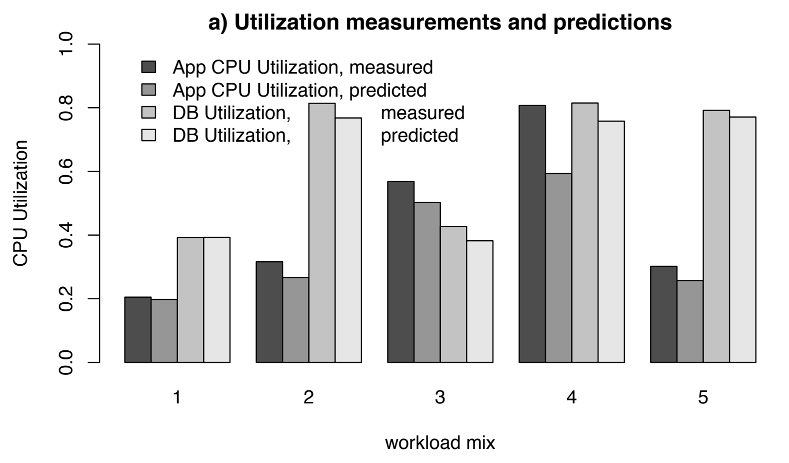
	Workload Intensities		
Mix	Task Management	Integration	AppBench
1	low	low	low
2	low	high	low
3	low	low	medium
4	low	high	high
5	medium	medium	low

Table 7.4: Scenario 4: Different Workload Mixes

Figure 7.16 a) shows the measured and predicted server CPU utilization for the different workload mixes. The utilization of both the database server and application servers vary from 20% to 80%. The utilization predictions for the database server fit the measurements quite well. The highest deviation of 7% can be observed for workload mix 4. The utilization predictions for the application servers have a prediction error comparable to the prediction error observed in Scenario 3. In workload mix 4, where the AppBench workload runs at high load, the relative prediction error is more than 25%. However, in workload mixes where the AppBench workload runs at low to medium load, the relative prediction error for the application server utilization is within 15%.

Figure 7.16 b) shows the relative error of the average response time predictions for the five workload mixes. The relative prediction error for a workload type is the mean relative prediction error of the workload type's services. For example, a relative prediction error of 10% for the task management workload means that the mean relative error of the average response time predictions of services *MyActivities*, *MyTeamsOpenActivities*, *MyTeamsClosedActivities* and *AllActivities* is 10%. For workload mixes 1, 2, 3 and 5, the prediction error is mostly below 20%. Only in workload mix 3, where the application server utilization is 57%, the AppBench requests have a higher prediction error of 40%. Considering workload mix 4, the prediction error of all workload types is higher than 30%. Apparently, the highly utilized application server has a significant impact on all requests, resulting in a significant slowdown. Whenever the application server is utilized below 40%, the predictions fit the measurements accurately. This is also reflected in the production environments of the SaaS provider: For the application servers, a utilization of more than 40% is tried to be avoided, given that the performance may degrade significantly for higher utilization levels. Since the application server tier is kept stateless, adding or removing application servers is easily possible during system operation without interruption.

Overall, the database scales very well, meaning that the utilization of the database grows almost linearly with the load intensity and the service response times increase slowly from low to high load levels. The CPU waiting time increases only slowly because of the high number of CPU cores. Apart from the CPU contention, another reason for the response time growth is contention for writing database logs. This contention is illustrated in Tables 7.5 and 7.6. Each table shows an excerpt of an Oracle Automatic Workload Repository (AWR) report (Kyte, 2005). Table 7.5 shows the top five wait classes and events for a request, obtained during a scenario of low load. Table 7.6 shows the same wait statistics obtained during a scenario where the load intensity is four times the load intensity as in the low load scenario. The most important wait classes, ranked by total wait time, are CPU time, system I/O and commit. A major factor for the commit waiting



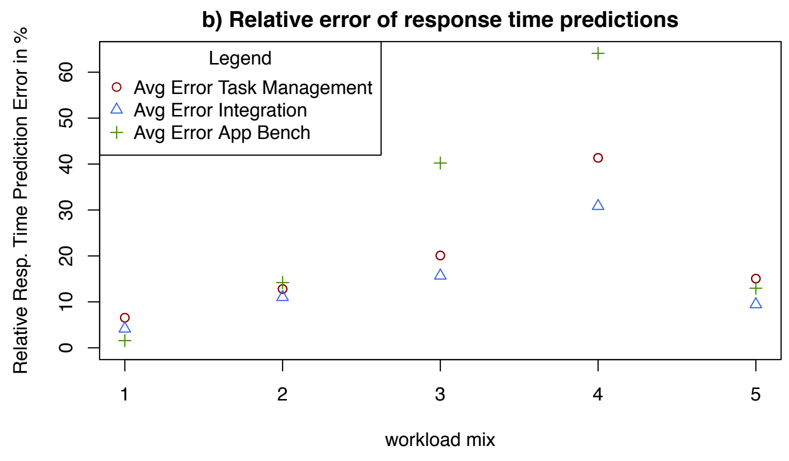


Figure 7.16: Scenario 4: Measurements, Prediction Results and Prediction Errors

time is the wait event *log file sync*. In the low load scenario, both commit and log file sync exhibit a total wait time of 235 seconds. In the AWR report where the load intensity has been quadrupled (Table 7.6), the important wait classes have a total wait time that is also approximately quadrupled. For instance, system I/O has a total wait time of 297 seconds in the low load scenario, and a total wait time of 1113 seconds in the high load scenario. However, the total wait time due to commit increases non-linearly with the load intensity. While it is 235 seconds in the low load scenario, it is 2071 seconds in the high load scenario, dominated again by wait event log file sync. Hence, for the considered workload, syncing Oracle's redo logs is an important factor and can become a bottleneck when scaling the intensity. This explains the moderate response time growth in high load scenarios besides the increased CPU contention.

Wait Class	Waits	Total Wait Time [s]	Average Wait [ms]
CPU time		14754	
System I/O	183560	297	2
Commit	83464	**235**	3
User I/O	17600	9	1
Network	5800062	9	0

Event	Waits	Total Wait Time [s]	Average Wait [ms]
log file sync	83464	**235**	3
log file parallel write	84557	150	2
db file parallel write	74425	107	1
log file sequential read	4136	30	7

Table 7.5: Oracle Automatic Workload Repository (AWR) Report, Low Load

Wait Class	Waits	Total Wait Time [s]	Average Wait [ms]
CPU time		63429	
Commit	341870	**2071**	6
System I/O	570897	1113	2
Concurrency	173645	117	1
Other	40087	73	2

Event	Waits	Total Wait Time [s]	Average Wait [ms]
log file sync	341870	**2071**	6
log file parallel write	282797	553	2
db file parallel write	225846	390	2
log file sequential read	22204	174	8

Table 7.6: Oracle Automatic Workload Repository (AWR) Report, High Load

7.3.5 Discussion

The SaaS provider case study demonstrates the modeling and prediction capabilities of our approach in the context of a real-life enterprise software system. The proposed modeling abstractions allow describing the performance-relevant factors of the component-based multi-tenant platform. The concept of probabilistic parameter dependencies proved to be useful to strike a balance between the model's abstraction level and its prediction accuracy. The prediction accuracy is investigated under different workload types, different workload intensities and different workload mixes. The achieved accuracy for the database utilization predictions is within 5% error. For the application server tier, the utilization predictions are accurate as long as the application servers operate at low to medium load levels. Higher utilization is avoided in the production environment because the system performance may degrade significantly due to the garbage collection overhead. For the service response times, the relative prediction error is mostly within 20%. This applies both to average service response times as well as to the 90th percentile response times. Thus, even without using modeling abstractions at a high level of detail, the response time distributions are captured in a representative way. The applied methods do not require monitoring capabilities that go beyond the monitoring infrastructure running in production.

7.4 SPECjEnterprise2010 Case Study

In our second case study, we apply the approach presented in this thesis in the context of a representative platform for distributed component-based applications, namely the Java EE platform, one of today's major technology platforms for building enterprise systems. Java EE provides a framework for building distributed web applications. Amongst others, it includes a server-side framework for component-based applications, the Enterprise JavaBean (EJB) architecture.

As application to evaluate, we chose the SPECjEnterprise2010 benchmark application described in Section 7.4.1. The case study covers all evaluation questions mentioned in Section 7.1. We investigate multiple application scenarios: Section 7.4.2 presents how we semi-automatically extracted an architecture-level performance model of SPECjEnterprise2010. Section 7.4.3 evaluates certain service behavior abstractions and probabilistic parameter dependencies of the benchmark application. Section 7.4.4 outlines a joint work together with Nikolaus Huber, where the extracted performance models are used as a basis for implementing an automatic performance-aware resource management approach (Huber et al., 2011a; Huber, 2014).

7.4.1 SPECjEnterprise2010 Benchmark Application

We selected the SPECjEnterprise2010 benchmark application as a basis for our case study since it models a representative state-of-the-art scalable system. It represents a classical multi-tier business information system with an application server tier and a database server tier. The benchmark is open source, thus it provides flexible instrumentation and deployment options, in contrast to the real-world application considered in Section 7.3. Previous versions of the benchmark have already been successfully applied for research purposes (Kounev, 2006; Kounev and Buchmann, 2003).

SPECjEnterprise2010 is a Java EE benchmark developed by SPEC's Java subcommittee for measuring the performance and scalability of Java EE based application servers. The benchmark workload is generated by an application that is modeled after an automobile manufacturer. As business scenarios, the application comprises customer relationship management (CRM), manufacturing, and supply chain management (SCM). The business logic is divided into three domains: orders domain, manufacturing domain, and supplier domain. To give an example of the business logic implemented by the benchmark, consider a car dealer that places a large order with the automobile manufacturer. The large order is sent to the manufacturing domain, which schedules a work order to manufacture the ordered vehicles. In case some parts needed for the production of the vehicles are depleted, a request to order new parts is sent to the supplier domain. The supplier domain selects a supplier and places a purchase order. When the ordered parts are delivered, the supplier domain contacts the manufacturing domain and the inventory is updated. Finally, upon completion of the work order, the orders domain is notified.

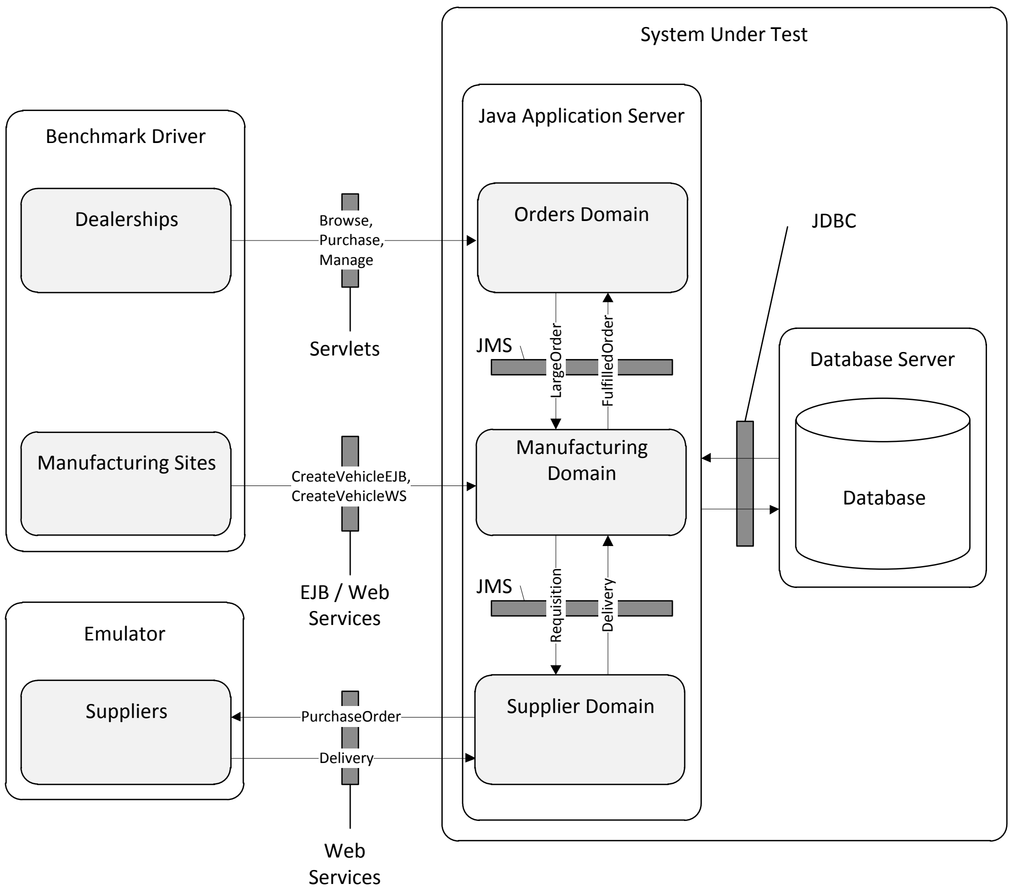


Figure 7.17: SPECjEnterprise2010 Architecture, cf. SPEC (2010)

Figure 7.17 depicts the architecture of the benchmark as described in the benchmark documentation. The benchmark application is divided into three domains: orders domain, manufacturing domain and supplier domain.The application logic in the three domains is implemented using EJBs which are deployed on the considered Java EE application server. The domains interact with a database server via Java Database Connectivity (JDBC) using the Java Persistence API (JPA). The communication between the domains is asynchronous and implemented using point-to-point messaging provided by the Java Message Service (JMS). The workload of the orders domain is triggered by dealerships whereas the workload of the manufacturing domain is triggered by manufacturing sites. Both, dealerships and manufacturing sites are emulated by the benchmark driver, a separate supplier emulator is used to emulate external suppliers. The communication with the suppliers is implemented using Web Services. While the orders domain is accessed through Java Servlets, the manufacturing domain can be accessed either through Web Services or EJB calls, i.e., Remote Method Invocation (RMI). As shown on the diagram, the system under test spans both the Java application server and the database server. The emulator and the benchmark driver have to run outside the system under test so that they do not affect the benchmark results. The benchmark driver executes five benchmark operations. A dealer may *browse* through the catalog of cars, *purchase* cars, or *manage* his dealership inventory, i.e., sell cars or cancel orders. In the manufacturing domain, work orders for manufacturing vehicles are placed, triggered either per WebService or RMI calls (*createVehicleWS* or *createVehicleEJB*).

7.4.2 Semi-Automatic Model Extraction and Model Parameterization

This section investigates the evaluation question of whether the semi-automatic model extraction and parameter estimation techniques described in Chapter 6 provide representative models, i.e., if the predictions based on the extracted models have acceptable accuracy. First, we describe how the extraction method is implemented and processed and how the SPECjEnterprise2010 benchmark is deployed in our experimental environment. We then illustrate the extracted model instance and analyze its performance prediction accuracy in various evaluation scenarios (for details see also Brosig et al. (2011)).

Context and Experiment Setup

As an application server implementing the Java EE specifications, we employ the Oracle WebLogic Server (WLS). Oracle WebLogic Server (WLS) comes with an integrated monitoring platform, namely the WebLogic Diagnostics Framework (WLDF). WLDF is a monitoring and diagnostics framework that enables collecting and analyzing diagnostic data for a running WLS instance. The two main WLDF features that we make use of are the data harvester and the instrumentation engine.

The data harvester can be configured to collect detailed diagnostic information about a running WLS and the applications deployed thereon. The instrumentation engine allows injecting diagnostic actions in the server or application code at defined locations. In short, a location can be the beginning or end of a method, or before or after a method call. Depending on the configured diagnostic actions, each time a specific location is reached during processing, an event record is generated. Besides information about, e.g., the time when or the location where the event occurred, an event record contains a *diagnostic context id*, which uniquely identifies the request that generated the event and allows to trace individual requests as they traverse the system. If a request occurs within a database transaction, an event record additionally supplies a *transaction id* (different from the diagnostic context id). Furthermore, the event records are identified with an auto-incrementing *event record id*. Other features justifying our selection of WLDF as monitoring tool are listed in the following: i) WLDF monitors that are injected during instrumentation can be enabled/disabled at run-time. ii) The amount of incoming requests that are traced can be throttled. The monitoring frequency can be constrained by a throttle interval or a throttle rate. iii) WLDF offers a monitoring action that does not generate any event records but makes response time measurements and aggregates them on-the-fly to metrics such as the mean or standard deviation. Since the aggregated data is stored in memory, the overhead of this action is very low.

We implemented the semi-automatic model extraction and model parameterization methods described in Chapter 6 to support EJBs, Java Servlets, and Java Server Pages (JSP), as well as Message-Driven Beans (MDBs) for asynchronous point-to-point communication using Java Message Service (JMS). In the following, we briefly describe the implementation of the extraction process.

For the extraction of the component connections according to Section 6.2.1, the component boundaries can be specified as groups of EJBs, Servlets and JSPs. Thus, WLDF is configured to monitor entries/exits of EJB business methods, Servlet services (including also JSPs) and JMS send/receive methods. As depicted in Figure 7.18, the WLDF diagnostic context id uniquely identifies a request, but forked threads receive the same diagnostic context id. Hence, to separate the different call paths from each other, the diagnostic context id is not sufficient. In those cases, we make use of the transaction id. The ordering of the event records is done via the event record id. Based on the set of observed call paths,

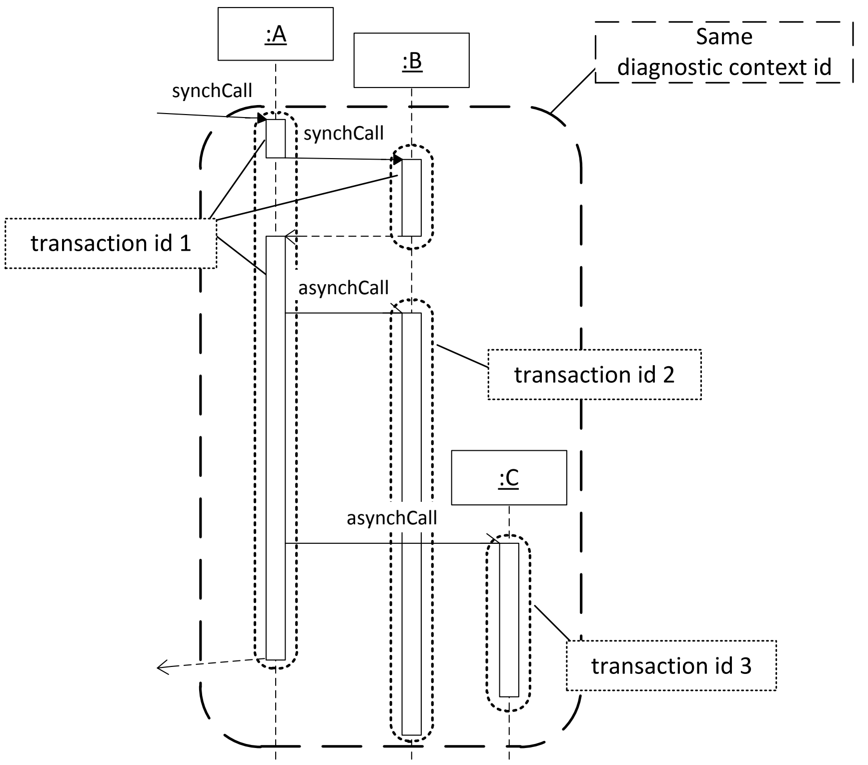


Figure 7.18: Diagnostic Context Id and Transaction Ids During Asynchronous Messaging

the effective connections among components can be determined, i.e., required interfaces of components can be bound to components providing the respective services.

For the extraction of the fine-grained behavior abstractions according to Section 6.2.2, we follow the approach described in Brosig et al. (2009). Performance-relevant actions are made explicit by method refactorings. This is because of the lack of tool support for in-method instrumentation. Current instrumentation tools including WLDF support only method-level instrumentation. They do not support instrumentation at custom locations other than method entries and exits. Control flow statistics are collected in parallel to the extraction of the fine-grained behavior abstractions. The sending of asynchronous JMS messages is modeled as fork action.

For the resource demand estimation of individual internal actions, we apply two approaches: i) in phases of low load we approximate resource demands with measured response times, ii) in phases of medium to high load we estimate resource demands based on measured utilization and throughput data with weighted response time ratios (see Section 6.4.3).

For the evaluation, we applied the implemented semi-automatic model extraction method to SPECjEnterprise2010. The resource environment is described in the following, before describing the extracted architecture-level performance model. We then compare the performance predictions provided by the extracted models with measurements on the real system considering different execution scenarios.

Resource Environment

We installed the benchmark in the system environment depicted in Figure 7.19. The benchmark application is deployed in an Oracle WebLogic Server (WLS) 10.3.3 cluster of up to eight physical nodes. Each WLS instance runs on a 2-core Intel CPU with OpenSuse 11.1. As a Database Server (DBS), we used Oracle Database 11g, running on a Dell PowerEdge R904 with four 6-core AMD CPUs, 128 GB of main memory, and 8x146 GB SAS disk drives. The benchmark driver master, multiple driver agents, the supplier emulator and the DNS load balancer were running on separate virtualized blade servers. As operating system, the virtual machines execute CentOS 5.3 and are equipped with two respectively four virtual CPUs. The blade servers are equipped with two 4-core Intel CPUs and 32 GB of main memory. The machines are connected by a 1 GBit LAN, the DBS is connected with 4 x 1 GBit ports.

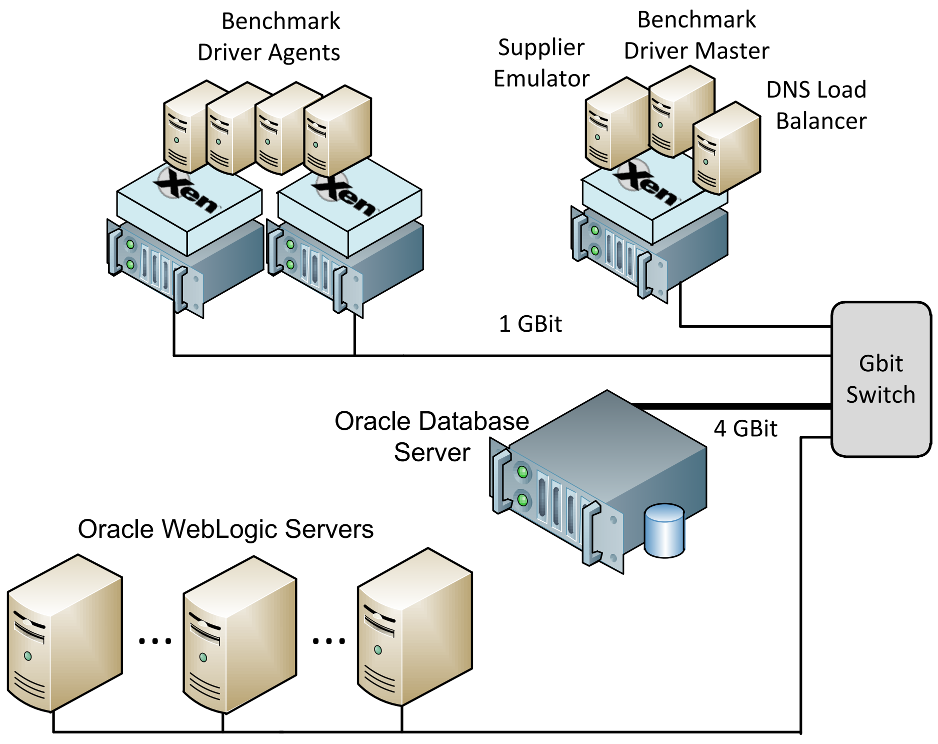


Figure 7.19: Experimental Environment for Semi-Automatic Model Extraction

Extracted Architecture-Level Performance Model

We consider the entire benchmark application, i.e., including supplier domain, dealer domain, the web tier and the asynchronous communication between the three domains. We deal with EJBs including MDBs for asynchronous point-to-point communication, web services, Servlets and JSPs. Figure 7.20 depicts a high-level overview of the basic structure of the extracted architecture-level performance model. The system model configuration shows a load balancer that distributes incoming requests to replicas of the SPECjEnterprise2010 benchmark application which themselves need an emulator instance and a database instance. A benchmark application instance refers to a composite component located in the component repository. The composite component in turn consists of component instances, e.g., a *SpecAppServlet* component or a *PurchaseOrderMDB* component. These components reside in the repository as well. The performance model of the benchmark application consists of 28 components whose services are described by 63 service behavior abstractions. In total, 51 internal actions, 41 branch actions and four loop actions have been modeled.

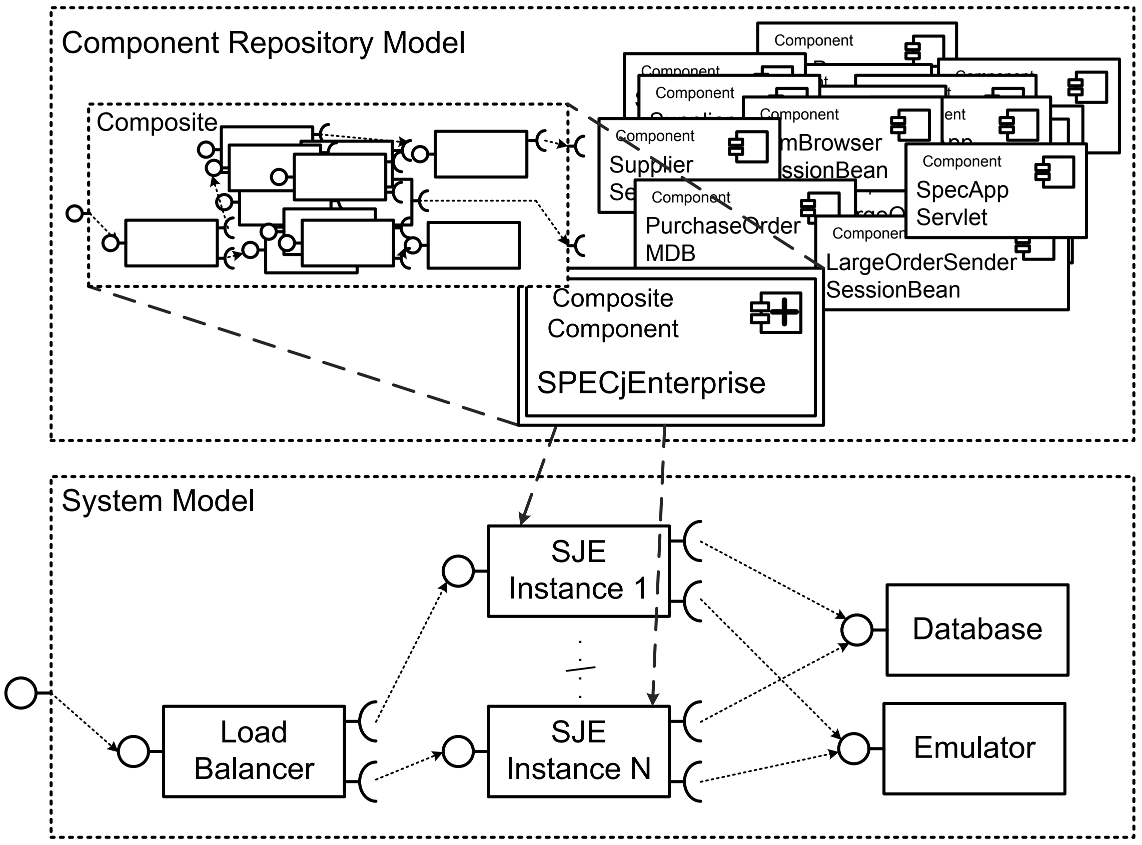


Figure 7.20: SPECjEnterprise2010 Model Structure

The resources we considered were the CPUs of the WLS instances (WLS CPU) and the CPUs of the database server (DBS CPU). The network and hard disk I/O load could be neglected. Note that we configured load-balancing via a Domain Name System (DNS) server. The DNS server is only used at benchmark start-up.

In order to apportion CPU resource demands among the database server (DBS) and the application server (WLS), we apply the following approximation: Looking at an EJB transaction, the transaction consists of a working phase and a commit phase. Processing times in the working phase are apportioned to the WLS CPU. Processing times in the commit phase are apportioned to the DBS CPU. The assumption is that the length of the working phase is proportional to the WLS CPU resource demand whereas the length of the commit phase is proportional to the DBS CPU resource demand. The approximation implies that, when estimating resource demands for the DBS CPU, we overestimate database writes and ignore database reads. However, given that Java Persistence API (JPA) implements internal entity caches that reside in the WLS instance, database reads are not expected to dominate the overall application performance. While processing times of working phases can normally be measured directly, for container-managed transactions, processing times of commit phases are not accessible with WLDF. To measure the length of the commit phase of a transaction, we compute the difference between method response times *inside* the method and response times *outside* the method. For the former case, we introduce time sensors after method entry and before method exit. For the latter case, the time sensors are placed around the method call at the callee. In this way, we obtain the approximate processing times of transaction commit phases.

To keep the overhead low, we separated the extraction step in which call paths are monitored from the step in which resource demands are estimated. Both steps were conducted with one single WLS instance. For the resource demand estimation, we use a WLDF method instrumentation feature where individual measurements are aggregated online to obtain statistics like, e.g., method invocation counts or average method response times. Given that individual measurements are not required to be stored, the overhead of this approach is an order of magnitude lower than the WLDF event record triggering approach.

The resource demands were extracted during a benchmark run with a steady state time of 900 seconds and a WLS CPU utilization of about 60%. The same benchmark run was then executed without any instrumentation in order to quantify the instrumentation overhead factor and adjust the estimates of the WLS CPU resource demands. Furthermore, we measured the delay for establishing a connection to the WLS instance which is dependent on the system load. With the knowledge of the number of connections the individual benchmark operations trigger, the load-dependent delay is estimated and taken into account in the predicted response times.

Results

The extracted models are evaluated by comparing the model predictions with measurements on the real system. The usage profile representing the benchmark workload has been provided manually. The five benchmark operations are modeled as individual usage scenarios.

In the investigated scenarios, we vary the throughputs of the dealer and manufacturing driver as well as the deployment configuration. Note that we extracted the performance model on a single WLS instance whereas for validation, WLS clusters of different sizes were used. As performance metrics, we considered the average response times of the five benchmark operations as well as the average utilization of the WLS CPU and the DBS CPU. In the cluster scenarios where several WLS instances are involved, we considered the average utilization of all WLS cluster node CPUs. Note that the response times of the benchmark operations are measured at the driver agents, i.e., the WLS instances run without any instrumentation. For each scenario, we first predicted the performance metrics for low load conditions ($\approx$ 20% WLS CPU utilization), medium load conditions ($\approx$ 40%), high load conditions ($\approx$ 60%), and very high load conditions ($\approx$ 80%), and then compared them with steady-state measurements on the real system.

Scenario 1: Cluster of two application server instances. For the first validation scenario, we configured an application server cluster consisting of two WLS instances and injected different workloads. The measured and predicted server utilization for the different load levels are depicted in Figure 7.21 a). The utilization varies from 20% to 80%. The utilization predictions fit the measurements very well, both for the WLS instances as well as for the DBS. Figure 7.21 b) shows the response times of the five benchmark operations for the four load levels. The response times of the benchmark operations are in the range of 10ms to 70ms. As expected, the higher the load, the faster the response times grow. Compared to the other benchmark operations, *Browse* and *Purchase* have lower response times, while *CreateVehicleEJB* and *CreateVehicleWS* take most time. In Figure 7.21 c), the relative error of the response time predictions is shown. The error is mostly below 20%, only *Browse* has a higher error but still lower than 30%. The prediction accuracy of the latter increases with the load. This is because *Browse* has a rather small resource demand but includes a high number of round trips to the server translating in connection delays (15 on average).

Scenario 2: Cluster of four application server instances. In Scenario 2, we considered a four node WLS cluster again at four different load levels. Figure 7.22 a) shows the measurements and predictions of the server utilization. Again, the predictions are accurate. However, one can identify a small deviation of the DBS CPU utilization that is growing with the load.

Figure 7.22 b) depicts the relative response time prediction error for Scenario 2. Again, the relative error is mostly below 20%. For the same reasons already mentioned in Scenario 1, operation *Browse* stands out a little. However, its prediction error is still below 30%.

Scenario 3: Cluster of eight application server instances. For Scenario 3, we deployed the benchmark in a cluster of eight WLS instances. As shown in Figure 7.23, the server utilization predictions are very accurate. While the DBS utilization prediction exhibits an error that grows with the injected load, it does not exceed 15%.

In cases of low to medium load, the accuracy of the predicted response times is comparable to Scenarios 1 and 2 (Figure 7.23 b). However, in cases of high load, the prediction error grows by an order of magnitude. This is because under this load level some of the WLS cluster nodes were overloaded, i.e., the cluster was not load-balanced anymore. The overloaded WLS instances then lead to biased response time averages. We assume that the cluster is unbalanced due to DNS caching effects that are not reflected in the performance model. These effects could not be observed in the setting with a single application server node in which the model was extracted.

Summary

We evaluated the performance model that was extracted with the methods described in Chapter 6 in various realistically-sized deployment environments under different workload mixes and load intensities. Concerning the CPU utilization, the observed prediction error for the WLS application server was below 5%. For the database server, the CPU utilization prediction error was mostly below 10%. The response time predictions of the benchmark operations mostly had an error of 10% to 20%. In the case of the eight node application server cluster, the response time predictions were not accurate for higher loads. This is because the cluster was not load-balanced correctly anymore. Nevertheless, the semi-automatic model extraction and parameter estimation techniques provided a performance model that is representative for most evaluation scenarios.

7.4.3 Service Behavior Abstractions and Probabilistic Parameter Dependencies

In the previous section, only fine-grained behavior abstractions were considered. In this section, using exemplary services of the SPECjEnterprise2010 benchmark, we: (i) show how behavior descriptions at different levels of abstraction can affect the performance

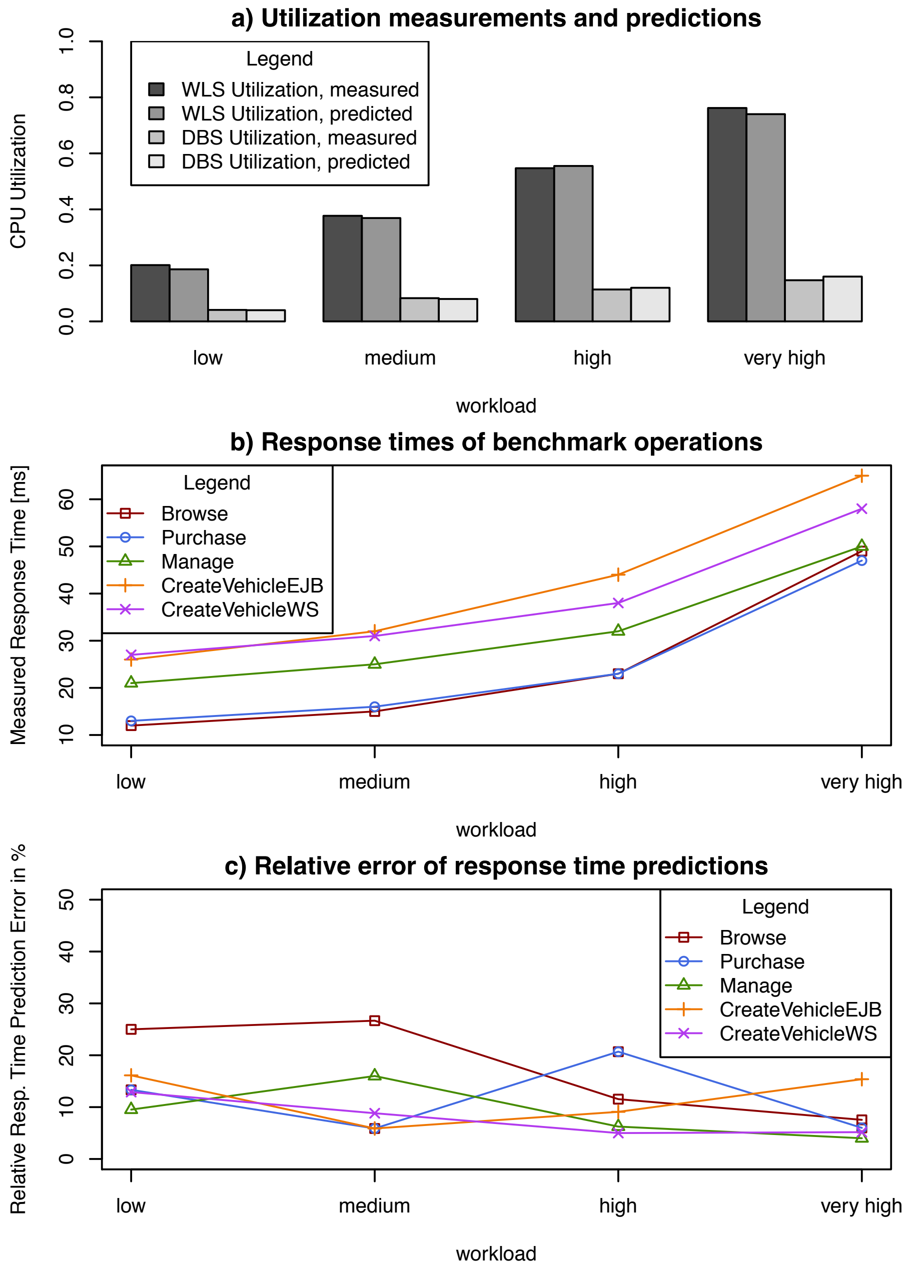


Figure 7.21: Scenario 1: Measurements, Prediction Results and Prediction Errors

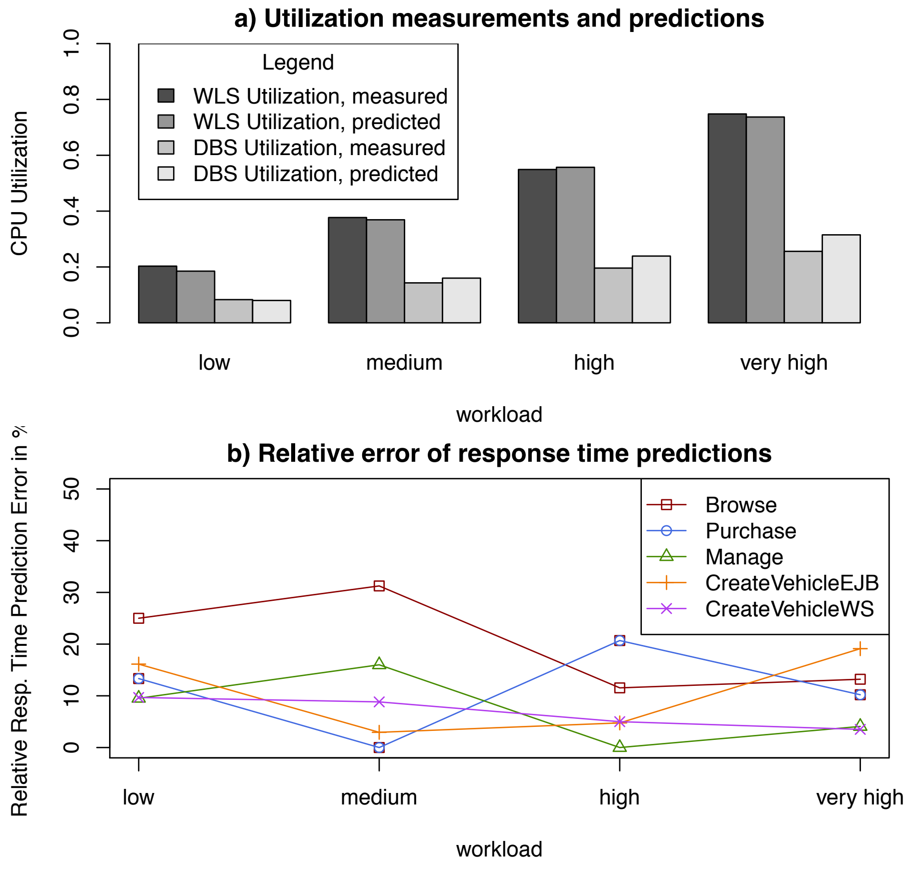


Figure 7.22: Scenario 2: Utilization Measurements and Relative Errors

prediction accuracy, and (ii) investigate a probabilistic parameter dependency by means of a sensitivity analysis (Brosig et al., 2013b, 2012).

For the measurement experiments, we use the same resource environment as in the previous section. In this section, the benchmark application runs on one WLS instance.

Coarse-Grained Behavior versus Fine-Grained Behavior

In SPECjEnterprise2010, the supplier domain has a component *PurchaseOrder*, as shown in Figure 7.24. It provides a service named *sendPurchaseOrder* that is responsible for dispatching the purchase orders. The sending operation supports two modes of operation: i) sending the order as an inline message without attachments, or ii) sending the order as a message with attachment. In the benchmark application, the probability of an inline message or a message with attachment each equals 0.5.

A fine-grained model of service *sendPurchaseOrder* is depicted in Figure 7.25. Internal service behavior is taken into account by reflecting the service's internal control flow. There is a branch action that either leads to an external service call to *processPurchaseOrderAttachment* or an external service call to *processPurchaseOrderInline*, both with a probability of 0.5.

A coarse-grained model of service *sendPurchaseOrder* is depicted in Figure 7.26. The external service calls to *processPurchaseOrderAttachment* and *processPurchaseOrderInline* are modeled as they can be observed from the component boundary of component *PurchaseOrder*. For each call to *sendPurchaseOrder*, a respective external service is either called once or not called at all. For both cases, one observes a probability of 0.5. In the model, this call frequency is described with the Probability Mass Function (PMF)

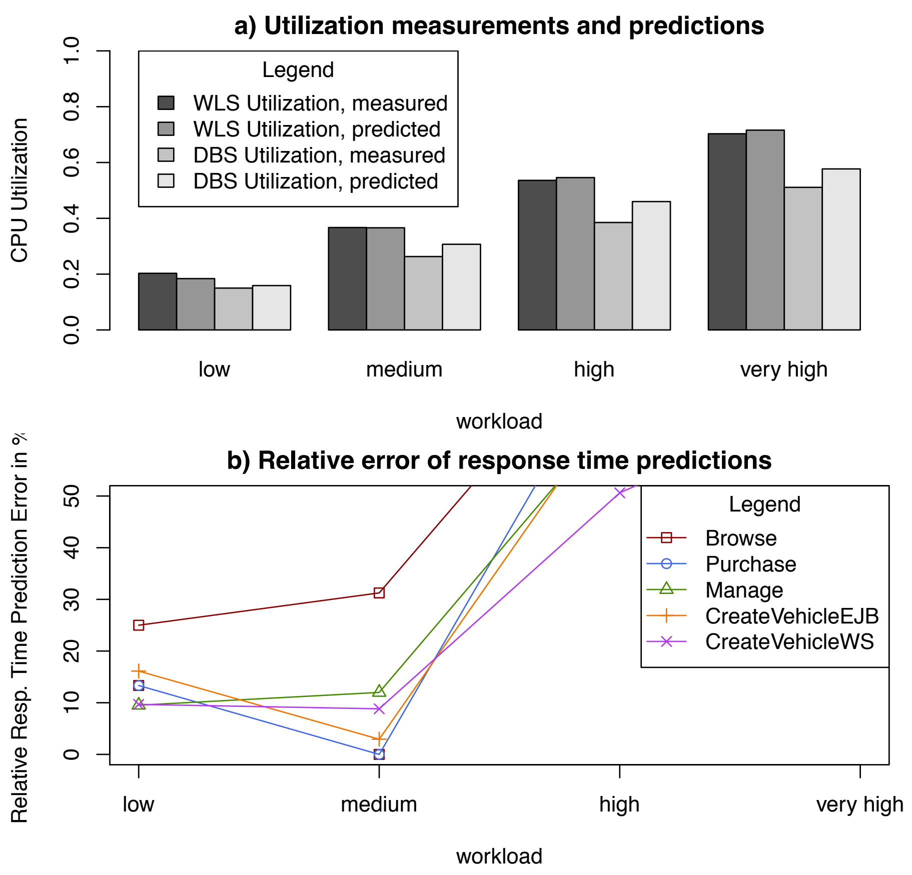


Figure 7.23: Scenario 3: Utilization Measurements and Predictions and Relative Error of Response Time Predictions

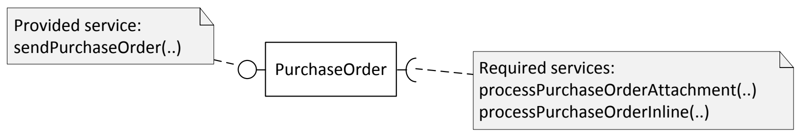


Figure 7.24: Component *PurchaseOrder*

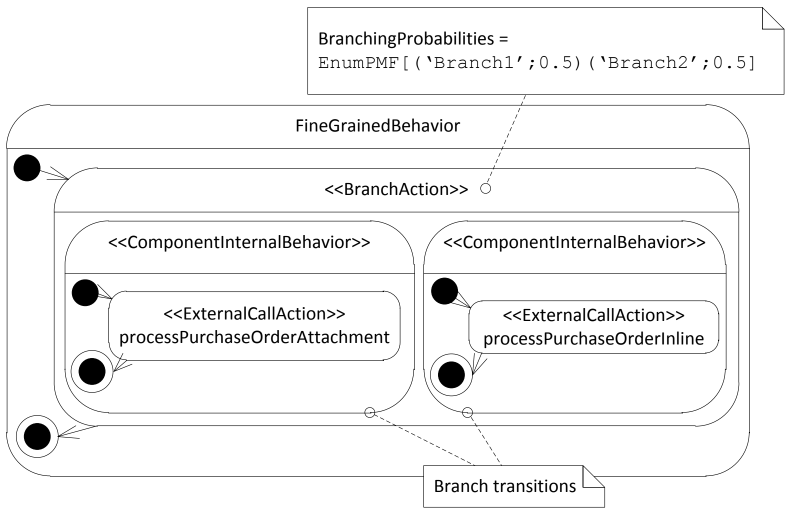

BranchingProbabilities =
EnumPMF[('Branch1';0.5)('Branch2';0.5]
FineGrainedBehavior
<<BranchAction>>
<<ComponentInternalBehavior>>
<<ComponentInternalBehavior>>
<<ExternalCallAction>>
processPurchaseOrderAttachment
<<ExternalCallAction>>
processPurchaseOrderInline
Branch transitions

Figure 7.25: Fine-Grained Behavior Abstraction of *PurchaseOrder#sendPurchaseOrder*

`IntPMF[(0;0.5)(1;0.5)]`. However, note that the exclusive relationship between the two external service calls cannot be reflected in the coarse-grained model.

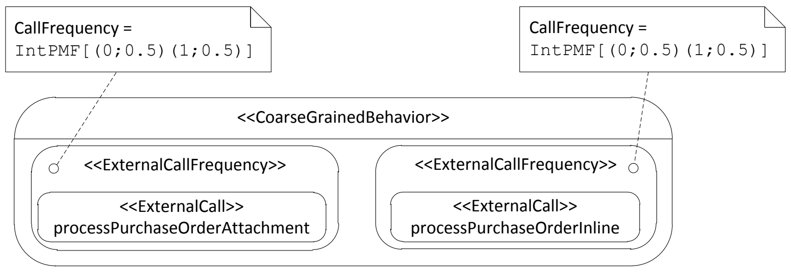

CallFrequency =
IntPMF[(0;0.5)(1;0.5)]
CallFrequency =
IntPMF[(0;0.5)(1;0.5)]
<<CoarseGrainedBehavior>>
<<ExternalCallFrequency>>
<<ExternalCallFrequency>>
<<ExternalCall>>
processPurchaseOrderAttachment
<<ExternalCall>>
processPurchaseOrderInline

Figure 7.26: Coarse-Grained Behavior Abstraction of *PurchaseOrder#sendPurchaseOrder*

Figure 7.27(c) shows measurements of the response time of *sendPurchaseOrder* as a histogram. The measurements were obtained during a benchmark run under medium load with a steady state time of 15 minutes. As expected, the measured response time distribution is multi-modal. We compare the measured response time distribution with predicted response time distributions using the fine-granular model (Figure 7.27(a)) and the coarse-grained model (Figure 7.27(b)). The resource demanding behavior of external service *processPurchaseOrderInline* was described as exponential service time with a mean of 10ms. Service *processPurchaseOrderAttachment* has a 30ms higher resource demand.

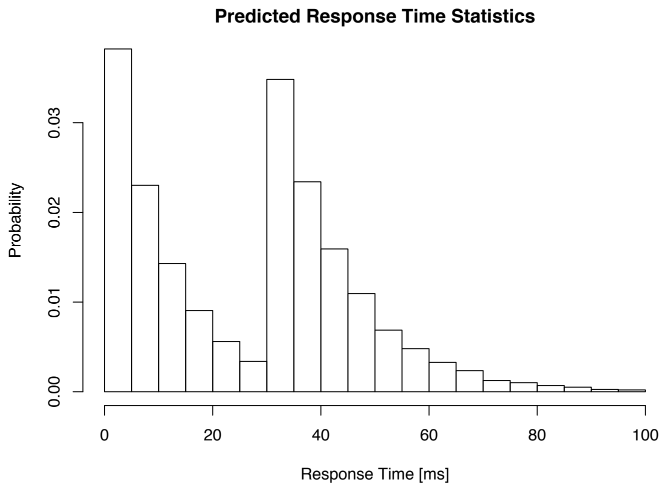
The fine-granular model reflects the bimodal distribution of *sendPurchaseOrder* better than the coarse-grained model. This is because the interdependency between the two external service calls is reflected in the fine-grained model through the branch action, while it is ignored in the coarse-grained model. The latter obviously affects the response time distribution, however, the mean values of the response time predictions do not differ. Both predictions indicate a mean of 26ms which well reflects the measured response time mean of 29ms. If one is only interested in predicting the mean response time, the coarse-grained abstraction in this case is sufficient. For a more representative response time distribution, the fine-grained abstraction is more suitable.

Probabilistic Parameter Dependency

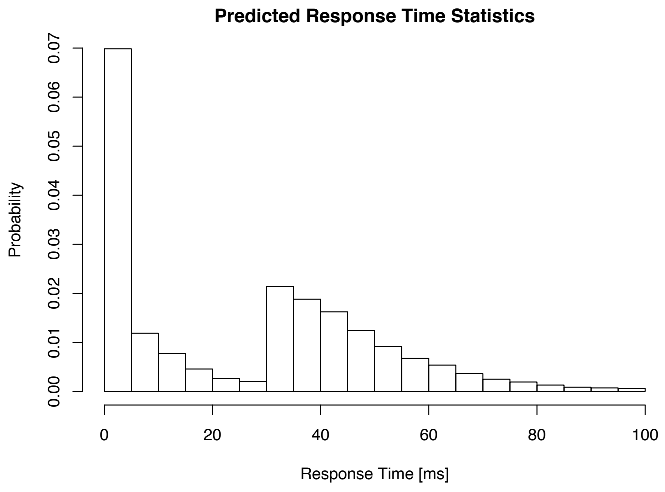
In the following, we analyze a probabilistic parameter dependency as it can be found in SPECjEnterprise2010. Figure 7.28 shows the *Manufacturing* component that provides a service *scheduleManufacturing* to schedule a new work order in the manufacturing domain for producing a set of assemblies. A work order consists of a list of assemblies to be manufactured and is identified with a *workOrderId*. In case the items needed to produce the assemblies available in the manufacturing site's warehouse are not enough, the *purchase* service of component *PurchaseOrder* is called to order additional items.

We are now interested in the probability of calling *purchase* which corresponds to a branch probability in the control flow of the *scheduleManufacturing* service. This probability will depend on the number of assemblies that have to be manufactured and the inventory of parts in the customer's warehouse. The higher the number of assemblies, the higher the probability of having to purchase additional parts.

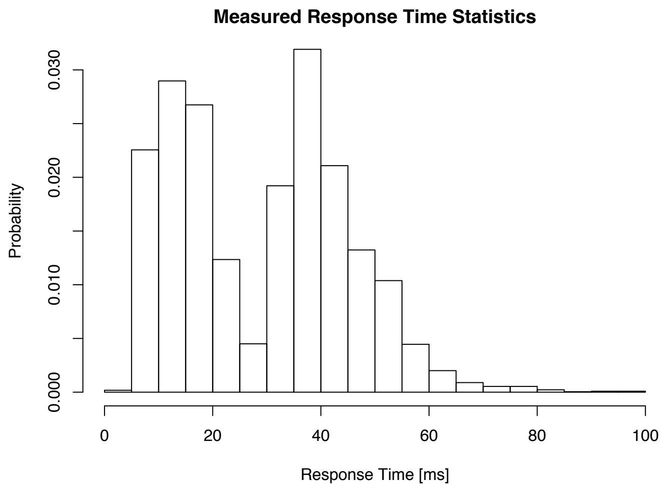
To better understand the considered dependency, in Figure 7.29 we show that the *Manufacturing* component is actually triggered by a separate *Dealer* component providing a *newOrder* service which calls the *scheduleManufacturing* service. The *newOrder* service receives as input parameters an *assemblyId* and *quantity* indicating a number of assemblies that are ordered by a dealer. This information is stored in the database in a data structure (see Figure 7.30) using *workOrderId* as a reference which is then passed to service *scheduleManufacturing* as an input parameter.



(a)



(b)



(c)

Figure 7.27: Response Time Statistics of *sendPurchaseOrder*: (a) Predicted using Fine-Grained Behavior Abstraction (b) Predicted using Coarse-Grained Behavior Abstraction (c) Measured

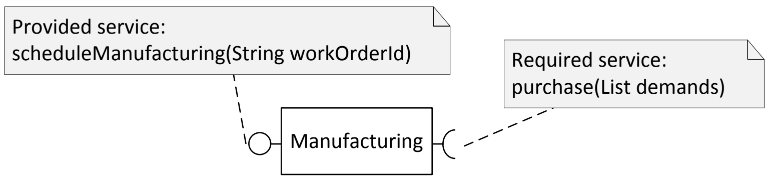


Figure 7.28: *Manufacturing* Component

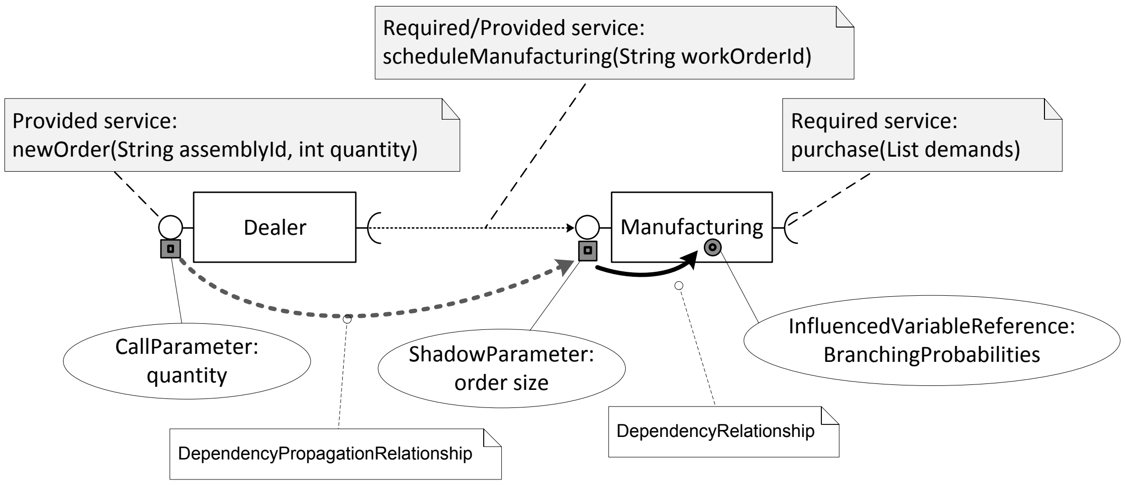

Required/Provided service:
scheduleManufacturing(String workOrderId)
Provided service:
newOrder(String assemblyId, int quantity)
Required service:
purchase(List demands)
Dealer
Manufacturing
CallParameter:
quantity
ShadowParameter:
order size
InfluencedVariableReference:
BranchingProbabilities
DependencyPropagationRelationship
DependencyRelationship

Figure 7.29: *Dealer* and *Manufacturing* Components

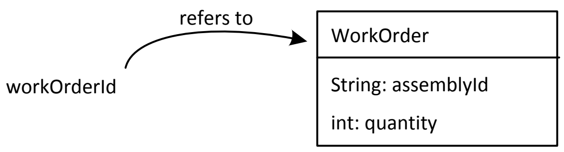


Figure 7.30: WorkOrder Data Structure

Intuitively, one would assume the existence of the following parameter dependency: The more assemblies are ordered (parameter *quantity* of service *newOrder* of the *Dealer* component), the higher the probability that new items will have to be purchased to refill stock (i.e., probability of calling *purchase* in the *Manufacturing* component). This dependency is modeled using probabilistic parameter dependencies as introduced in Section 4.1.5. We model a ShadowParameter named *order size*. This parameter characterizes the size of the incoming work order with id *workOrderId*. We add a dependency relationship between *order size* and the branching probabilities determining whether service *purchase* is called or not. Furthermore, we add a dependency propagation relationship between the call parameter *quantity* of service *newOrder* and the shadow parameter *order size* of the *Manufacturing* component. Knowing about the existence of the parameter dependency, we can use monitoring statistics collected at run-time to characterize the dependency probabilistically.

Figure 7.31 shows monitoring statistics that we collected at run-time showing the dependency between the influencing parameter *quantity* and the observed relative frequency of the *purchase* service calls. For instance, if the quantity equals to 20, in roughly one out of every four calls of *scheduleManufacturing*, service *purchase* was called.

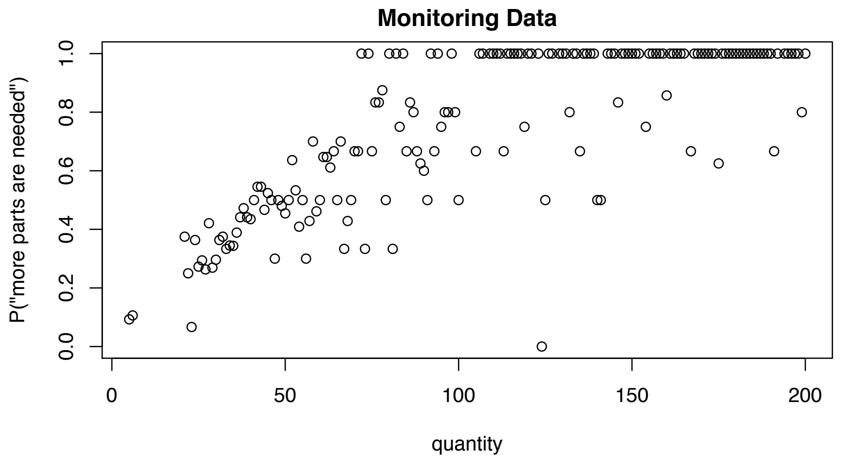


Figure 7.31: Measurement Statistics of *scheduleManufacturing* Service

Figure 7.32 shows how the dependency can be characterized probabilistically by considering three ranges of possible quantities. For example, for quantities between 50 and 100, the probability of a *purchase* call is estimated to be 0.67.

To illustrate the relevance of the considered parameter dependency, we conducted experiments with an adapted *scheduleManufacturing* workload.

- In the first scenario, we called *scheduleManufacturing* varying the quantity parameter at random in the range between 0 and 10.
- For the second scenario, we varied the quantity parameter between 0 and 200, while keeping the workload intensity at the same level as in the first scenario.

The measurements were obtained in experiment runs with a steady state time of 15 minutes. Knowing the monitoring statistics in Figure 7.31, we expect the number of *purchase* calls in the second scenario to be higher than in the first scenario. Given that *scheduleManufacturing* calls the *purchase* service asynchronously using point-to-point messaging provided by JMS, the more frequent *purchase* calls in the second experiment should not affect the response time of *scheduleManufacturing* directly but result in a higher application server utilization.

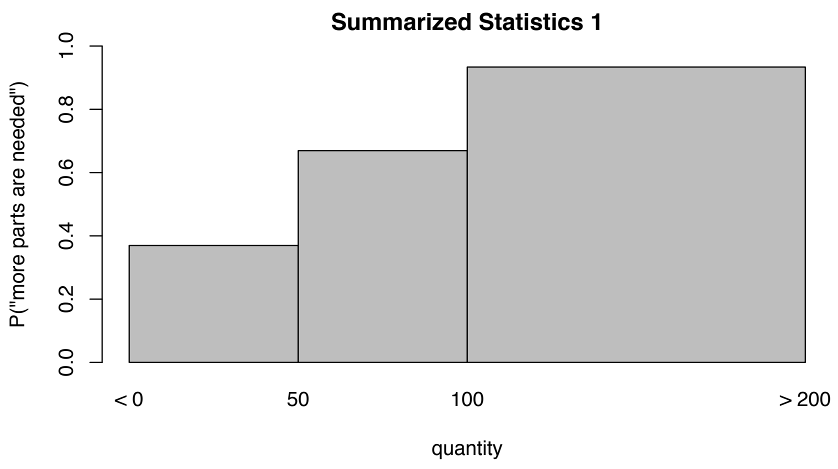


Figure 7.32: Aggregated Statistics of *scheduleManufacturing* Service

The results of the experiment runs are shown in Table 7.7. While in the first experiment the average utilization of the Oracle WebLogic Server (U_{WLS}) is approximately 17%, it increases to 63% in the second experiment. The increased mean response time (R_{avg}) of service *scheduleManufacturing* is explained by the increased utilization.

Workload:	Measurements		Predictions	
scheduleManufacturing	U_{WLS}	R_{avg} [ms]	U_{WLS}	R_{avg} [ms]
quantity in [0..10]	0.166	17.2	0.365	19.4
quantity in [0..200]	0.627	32.9	0.607	28.3

Table 7.7: Measurements and Predictions for the *scheduleManufacturing* Scenarios

Using a fine-grained behavior model with a characterization of the parameter dependency as shown in Figure 7.32, we made predictions for both scenarios. Regarding the second scenario, the utilization U_{WLS} and the average response time R_{avg} differ from the measurements with a small, negligible error. However, concerning the first scenario, with U_{WLS}=0.365 the prediction particularly overestimates the server utilization. The overestimation is a result of the aggregated monitoring statistics. For the interval between 0 to 50, the model simplifies the probability of a *purchase* call to be 0.37.

Adapting the characterization of the parameter dependency as shown in Figure 7.33, the prediction yields representative results: With U_{WLS}=0.196 and R_{avg}=16.8ms, the predictions match the measurements also in the first scenario.

The evaluation shows the performance-relevance of the parameter dependency between the influencing parameter *quantity* and the observed relative frequency of the *purchase* service calls. It further shows that detailed monitoring statistics are of importance for a representative characterization of probabilistic parameter dependencies.

Summary

We used scenarios from the SPECjEnterprise2010 benchmark application to evaluate if behavior descriptions at different levels of abstraction can be modeled, and how they affect the performance prediction accuracy. We showed that fine-grained models may reflect the response time distributions in a more representative way than coarse-grained models. If one is only interested in predicting the mean response time, coarse-grained abstractions are sufficient. Furthermore, we showed how to model parameter dependencies as they occur in

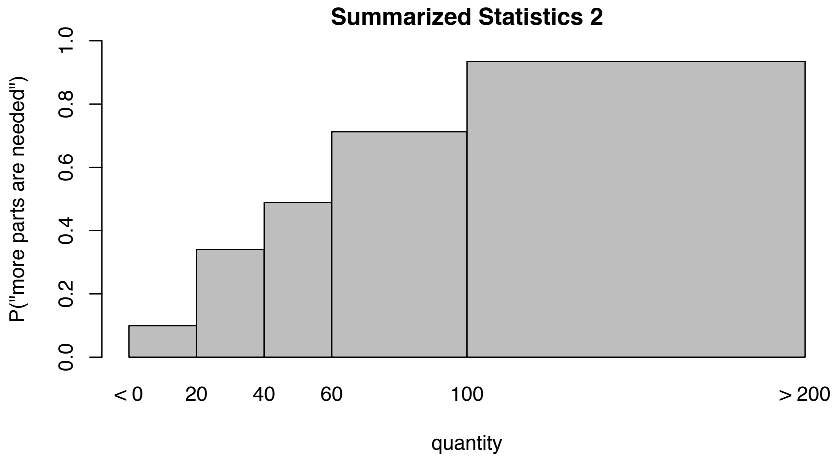


Figure 7.33: Aggregated Statistics of the *scheduleManufacturing* Service

modern enterprise software systems. We provided a sensitivity analysis for a major probabilistic parameter dependency within the manufacturing domain of SPECjEnterprise2010. Characterizing the probabilistic parameter dependency in a representative fashion allowed us to conduct accurate performance predictions for service input parameter variations.

7.4.4 Autonomic Performance-Aware Resource Management

We now use the extracted performance models of Section 7.4.2 and apply them in an end-to-end SPECjEnterprise2010 case study realizing an automatic performance-aware resource management approach (Huber et al., 2011a; Huber, 2014). The experiments have been conducted in collaboration with Nikolaus Huber.

Context and Experiment Setup

Highly variable workloads make it challenging to provide quality-of-service guarantees while at the same time ensuring efficient resource utilization (Kounev et al., 2010). To avoid violations of SLAs or inefficient resource usage, resource allocations have to be adapted continuously during operation to reflect changes in application workloads. For an end-to-end case study on autonomic performance-aware resource management, Huber et al. use the performance models of Section 7.4.2 to predict the effects of changes in user workloads, as well as to predict the effects of respective reconfiguration actions, performed to avoid SLA violations or inefficient resource usage.

Briefly, the resource allocation mechanism consists of two phases: a PUSH phase and a PULL phase. The PUSH phase allocates additional resources until all client SLAs are satisfied. The PULL phase optimizes the resource efficiency by deallocating resources that are not utilized efficiently (Huber et al., 2011a). The phases are conducted on the model level, i.e., without applying any changes on the real system until an adequate resource allocation is found.

Resource Environment

In contrast to the resource environment shown in Section 7.4.2, we use virtualized blade servers for the application server cluster. Each blade server is equipped with two Intel Xeon E5430 4-core CPUs running at 2.66 GHz and 32 GB of main memory. The machines are connected by a 1 GBit LAN. Figure 7.34 shows the resource environment. We run Xen Server as the virtualization layer. As operating system, the VMs execute CentOS 5.3.

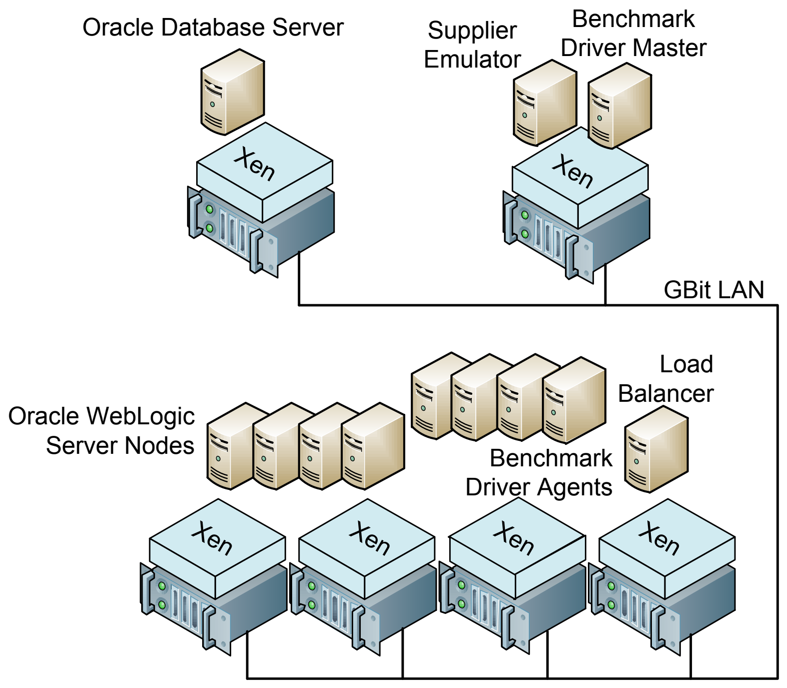


Figure 7.34: Experimental Environment for Autonomic Performance-Aware Resource Management, cf. Huber et al. (2011a)

The load balancer is haproxy 1.4.8 using round-robin as load balancing strategy, i.e., not a DNS load balancing as used in Section 7.4.2. The database is an Oracle 11g database server instance deployed on a Virtual Machine (VM) with eight virtual Central Processing Units (vCPUs) on a separate blade server running Windows Server 2008.

The SPECjEnterprise2010 benchmark application is deployed in a cluster of WLS nodes. For the evaluation, we considered reconfiguration options concerning the WLS cluster and the vCPUs the VMs are equipped with: WLS nodes are added to or removed from the WLS cluster, vCPUs are added to or removed from a VM. These reconfigurations are applicable at run-time, i.e., can be applied while the benchmark application is running.

Results

Several evaluation scenarios are investigated. The resource management approach is evaluated if it continuously keeps the system in a state such that SLAs are satisfied and resources are utilized efficiently.

Scenario 1: Adding a new service. The first scenario is intended to evaluate the results of the approach when a new service is deployed in the environment on-the-fly. Assume that there are four services executed in the environment running on one node with two vCPUs (denoted as default configuration c_0). The four running services are as follows: CreateVehicleEJB (abbreviated to EJB), Purchase, Manage and Browse. Their SLAs and their mean response times in configuration c_0 are depicted in Figure 7.35.

Now a new service (service CreateVehicleWS abbreviated as WS) with a corresponding SLA is added. To ensure that all SLAs are still maintained after deployment of the new service, the resource allocation mechanism is triggered. After adding the new service to the model, it predicts SLA violations for the services CreateVehicleWS and Purchase. Hence, the PUSH-Phase of the reconfiguration algorithm starts and suggests a capacity increase by one, adding an additional vCPU to the existing node (configuration c_1). After this

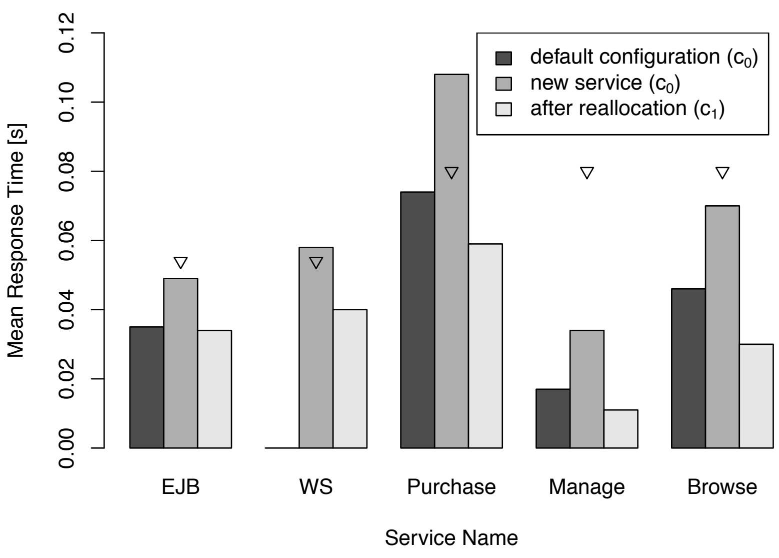


Figure 7.35: Scenario 1: Response Times and Respective SLAs (denoted by ▽) of the Five Benchmark Operations, cf. Huber et al. (2011a)

change, the simulation indicates satisfied SLAs, hence the algorithm enters the PULL-Phase and tries to reduce the overall amount of used resources considering all workload classes, but fails because then the SLAs of CreateVehicleWS and Purchase are again violated. Therefore, the resulting configuration proposed by the algorithm consists of one node with three vCPUs.

The above behavior was confirmed in our experiments depicted in Figure 7.35. The measurements show that with the default resource allocation the SLA for service CreateVehicleWS and Purchase cannot be sustained. However, after applying the resource allocation proposed by our algorithm, all SLAs are satisfied.

Note that we show an excerpt of the case study results here. For Scenario 2 and Scenario 3, where the resource allocation mechanism handles increasing user workload, respectively decreasing user workload, we refer to Huber et al. (2011a).

Scenario 4: Resource usage and efficiency. To show the benefits of the resource management approach, imagine a workload trend over seven days like the one depicted in Figure 7.36. In a static scenario, one would assign three dedicated servers to guarantee SLA compliance for the peak load. However, with the resource management approach one can dynamically assign the system resources. In the static scenario, one would use $7 \cdot 3 = 21$ servers, whereas the approach needs only $1 + 2 + 3 + 2 + 3 + 1 + 1 = 13$ servers. Hence, in such a scenario, only 62% of the resources of the static assignment are needed and thereby almost 40% of the available resources can be saved.

Summary

We explored the use of architecture-level performance models as a means for online performance prediction allowing to predict the effects of changes in user workloads, as well as to predict the effects of respective reconfiguration actions, performed to avoid SLA violations or inefficient resource usage. In this experiment, SPECjEnterprise2010 is deployed in a virtualized cluster environment. The case study serves as a proof-of-concept showing the feasibility of using architecture-level performance models at run-time and the benefits they provide.

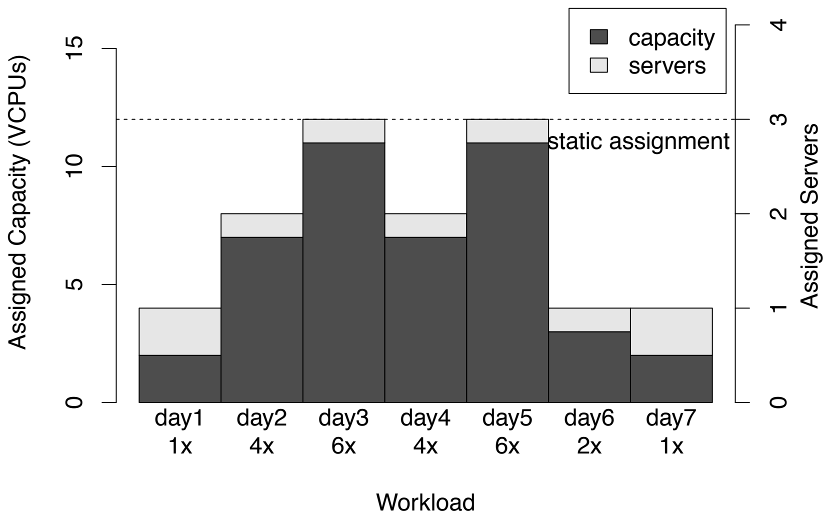


Figure 7.36: Assigned Capacity and Servers for an Exemplary Workload Trend, cf. Huber et al. (2011a)

7.4.5 Discussion

As part of the presented SPECjEnterprise2010 case study, all the evaluation goals established in Section 7.1 were considered. The proposed modeling abstractions allow describing the performance-relevant factors of the software architecture of SPECjEnterprise2010. Parameter dependencies can be modeled and probabilistically characterized. Furthermore, we successfully applied the semi-automatic model extraction and parameter estimation techniques that we presented in Chapter 6.

The attained prediction accuracy in various realistically-sized deployment environments under different workload mixes and load intensities was below 5% error for the application server CPU utilization, and mostly within 10% to 20% and not exceeding 30% error for response time predictions. The prediction capabilities served as input for an autonomic performance-aware resource management. Effects of changes in user workloads, effects of reconfiguration actions with the goal to avoid SLA violations or inefficient resource usage could be predicted with sufficient accuracy.

Given that the SPECjEnterprise2010 benchmark application is designed to be representative of enterprise software systems using Java EE, we consider the results of the case study to be externally valid for other enterprise software systems that implement a typical multi-tier architecture.

7.5 Summary

Section 1.4.1 identified several characteristics as essential success criteria for any model-based prediction approach. The evaluation goals presented in Section 7.1 are formulated with respect to these characteristics. The evaluation goals specifically address the modeling and prediction capabilities of the proposed modeling abstractions, model solving methods, and model maintenance methods.

For the evaluation in a realistic context, we selected two representative case studies. The first case study is a real-life enterprise software system from a large SaaS provider. The second case study uses the SPECjEnterprise2010 benchmark. It is a benchmark designed to serve as a representative application of today's enterprise Java systems.

In the SaaS provider case study, we demonstrated that our approach is applicable to component-based software systems of realistic size and complexity and that the modeling and prediction techniques provide sufficiently accurate performance predictions. The prediction accuracy is investigated under different workload types, different workload intensities, and different workload mixes. The attained accuracy for the database utilization predictions was within 5% error. For the application server tier, the utilization predictions were accurate as long as the application servers are operated at a low to medium load level. Higher utilization is avoided in production environments because the application performance may degrade significantly due to garbage collection overhead. For the service response times, the relative prediction error was mostly within 20%. This applies both to average service response times as well as to the 90th percentile response times, i.e., the response time distributions are captured in a representative way.

In contrast to the SaaS provider case study where the most stressed resource is the database server tier, in the SPECjEnterprise2010 case study we considered mostly application server-intensive workloads. We evaluated the proposed semi-automatic model extraction and parameter estimation methods. We investigated multiple application scenarios (see Section 1.5), including a scenario where the prediction capabilities were used as a basis for implementing an automatic performance-aware resource management approach. The achieved prediction accuracy in various realistically-sized deployment environments under different workload mixes and load intensity levels was below 5% error for the application server CPU utilization, and mostly within 10% to 20% and not exceeding 30% error for response time predictions. The effects of changes in user workloads as well as the effects of reconfiguration actions with the goal to avoid SLA violations or inefficient resource usage could be predicted with sufficient accuracy.

In this chapter, we demonstrated that: (i) the proposed performance abstractions lend themselves well to describe architecture-level performance models that are representative in terms of the performance properties of the modeled systems, (ii) that the proposed prediction mechanisms are capable of deriving performance predictions in online scenarios, and (iii) that the proposed model extraction and maintenance methods are suitable to extract and maintain performance model instances that provide an acceptable accuracy.

8. Concluding Remarks

This chapter provides a summary of the contributions presented in this thesis. Afterwards, we discuss future related research topics, particularly in the area of Quality of Service (QoS) prediction techniques at run-time.

8.1 Summary

This thesis proposed architecture-level performance models for online performance prediction. The ability to predict the performance impact of changes in the system environment as well as changes to the system configuration in an online setting provides the following important benefits:

- It is possible to anticipate which changes in the system environment, e.g., changing usage profiles or changing load intensities, would result in a performance degradation leading to Service Level Agreement (SLA) violations.
- It is possible to assess which changes to the system, e.g., resource reallocations, are suitable adaptation actions to ensure that the system's operational goals are maintained.

With our proposed modeling and prediction facilities, the above tasks can be performed on the model level *without* disturbing system operation, i.e., without having to change the running system in a trial-and-error approach. Performance questions about the impact of workload changes or system reconfigurations can be answered without running extensive performance tests in production-like test environments. Hence, our work provides a solid basis for developing model-based autonomic performance and resource management techniques that continuously adapt the system during operation in order to ensure that performance objectives are satisfied while at the same time system resources are used efficiently.

As pointed out in Chapter 3, existing approaches to online performance prediction normally abstract the system at a high level without explicitly taking into account its software architecture (e.g., flow of control and dependencies between software components) and configuration. Many restrictive assumptions are often imposed such as a single workload class, single-threaded components, homogeneous servers, or exponential request interarrival times and exponential service demands.

We developed **architecture-level performance abstractions for component-based software systems specifically designed for online use**. The modeling abstractions are part of the Descartes Modeling Language (DML), a new modeling language for run-time performance and resource management of modern dynamic IT service infrastructures.

The novel architecture-level performance abstractions involve: (i) a new approach to model performance-relevant service behavior at different levels of granularity (Section 4.1.3), (ii) a new approach to parameterize performance-relevant properties of software components (Section 4.1.4), and (iii) a new approach to model dependencies between parameters, each specifically designed for use at run-time (Section 4.1.5).

Service behavior can be modeled at different levels of abstraction and detail. The models are usable in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to accurate performance prediction. Furthermore, the modeled abstraction levels reflect the information that monitoring tools can obtain at run-time, e.g., to what extent component-internal information is available.

The service behavior models are parameterized with resource demands, response times, frequencies of external calls, loop iteration counts, and branching probabilities. In the context of online performance models, these parameters are typically characterized with probability distributions based on monitoring data collected at run-time. Our modeling abstractions allow specifying a so-called scope for model parameters that indicate if and how monitoring data for a given parameter, collected at the component instance level, can be aggregated with data collected at other component instances.

Furthermore, the modeling abstractions allow modeling relationships between parameters that are to be probabilistically characterized using monitoring data. The behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation. Moreover, the behavior of component services may be dependent on the state of data containers such as caches or on persistent data stored in a database. In many practical situations, providing an explicit characterization of such a dependency is not feasible since modeling the state of a cache and/or a database is extremely complex. This situation is common in business information systems and our modeling abstractions provide means to deal with it.

The contributions discussed above were published in Brosig et al. (2013b, 2012); Brosig (2011).

To enable online performance predictions with models described using the introduced modeling language, we provided a performance prediction process that strikes a balance between prediction accuracy and time-to-result. We provided a **performance prediction process that is tailored to a performance prediction request** (Chapter 5). To specify such a performance prediction request, referred to as a performance query, we introduced the Descartes Query Language (DQL) (Section 5.5). It is a language to express the demanded performance metrics for prediction as well as the specific goals and constraints in an online prediction scenario. In the context of online performance prediction, there are situations where the prediction results need to be available very fast to adapt the system *before* performance issues arise, and there are situations where a fine-grained prediction is used to find an efficient system configuration. However, an accurate fine-grained performance prediction comes at the cost of higher prediction overhead. By using more coarse-grained performance models one can speed up the prediction process.

Part of the prediction process is, on the one hand, the parameterization of involved model parameters with current up-to-date monitoring data, and on the other hand, the characterization of involved relationships between parameters, propagating the service input parameters specified as part of the usage profile. The result is a performance model instance where all model parameters (such as resource demands, response times, branching probabilities, call frequencies and loop iteration numbers) required to answer the given performance query are characterized with up-to-date monitoring data for a specific usage context.

Based on the given performance query, the performance prediction process then chooses a suitable model solving technique and abstraction level, and returns the performance metrics as requested. The prediction process uses existing model solving techniques based on established stochastic modeling formalisms, namely bounds analysis, a transformation to Layered Queueing Networks (LQNs), and a transformation to Queueing Petri Nets (QPNs) (Section 5.3). The prediction process decides which concrete model solving technique to apply. It also selects suitable configuration options of the applied model solving technique with the goal of tailoring the solution method to the given performance query.

DQL has been developed in the master's thesis of Gorsler (2013) and published in Gorsler et al. (2014, 2013). Note that the use of DQL is not restricted to DML, it can also be used as a generic interface to other performance prediction techniques. The investigation of the trade-off between prediction accuracy and time-to-result is published in Brosig et al. (2014).

For online performance predictions, it is essential to keep the performance model in sync with the modeled system. The model should provide up-to-date information about the system to enable accurate performance predictions. We thus proposed methods to ensure that the model constantly *mirrors* the performance-relevant structure and behavior of the system, that is, methods to **integrate architecture-level performance models and system environments** (Chapter 6). The integration has been realized by a technique to extract model instances semi-automatically based on monitoring data and a technique to automatically maintain the extracted instances at run-time. In each case, we distinguish between structural information about the system environment (e.g., involved software components) and model parameters (e.g., resource demands). Structural information is extracted via monitoring traces, model parameters are characterized based on monitoring data. At run-time, the system components are deployed in the target production environment. This makes it possible to obtain representative estimates of the various model parameters and to continuously adjust them to iteratively refine their accuracy.

We presented a classification of methods to obtain resource demand estimates (both in native and virtualized environments), we described methods to obtain response time distributions, loop iteration numbers, call frequencies and branch probabilities. Moreover, we described how relationships between model parameters can be characterized using monitoring data. However, in an online setting, monitoring has to be handled with care. Hence, we presented techniques to keep the monitoring overhead within limits such that system operation is not significantly disturbed.

Moreover, we presented a technique to calibrate and adjust an architecture-level performance model in order to increase its accuracy. In this thesis, calibrating a performance model means comparing the model predictions with measurements on the real system. If the deviation between performance predictions and measurements are observed to be systematic, improving the model accuracy is possible by automatically adjusting the model parameters without having detailed knowledge about the modeled system. The result of the adjustment is an increased prediction accuracy.

The contributions of Chapter 6 were published in Brosig et al. (2011, 2009, 2013a).

To validate the contributions of this thesis, we conducted multiple case studies (Chapter 7). We presented case studies with the SPECjEnterprise2010 benchmark, a benchmark designed to serve as a representative application of today's enterprise Java systems, as well as a case study with a real-life enterprise software system from a large Software-as-a-Service (SaaS) provider.

In the SPECjEnterprise2010 case study, the attained prediction accuracy in various realistically-sized deployment environments under different workload mixes and load intensities was below 5% error for resource utilization, and mostly within 10% to 20% and not exceeding 30% error for response time predictions which is considered acceptable for capacity management. The prediction capabilities were used as a basis for implementing an autonomic performance-aware resource management technique. The effects of changes in user workloads as well as the impacts of reconfiguration actions have been predicted with sufficient accuracy to avoid SLA violations or inefficient resource usage. In the SaaS provider case study, we investigated the prediction accuracy under different workload types, different workload intensities and different workload mixes. The attained prediction accuracy for resource utilization was within 5% error. For the service response times, the relative prediction error was mostly within 20%. This applies both to average service response times as well as to the 90th percentile response times, i.e., the response time distributions are also captured in a representative way.

The case studies demonstrated: i) that the proposed performance abstractions lend themselves well to describe architecture-level performance models that are representative in terms of the performance properties of the modeled systems, ii) that the proposed prediction mechanisms can be effectively used to derive performance predictions in online scenarios, and iii) that the proposed model extraction and maintenance methods are suitable to extract and maintain performance model instances that provide an acceptable accuracy.

The results of the evaluations were published in Brosig et al. (2013b, 2012, 2011); Huber et al. (2011a). The SaaS provider case study has not been published yet.

Our approach is the first approach to online performance prediction that uses architecture-level performance models. The performance prediction process facilitates flexible model-based predictions at run-time, combining the strength of simulative as well as analytical model solving techniques in a novel tailored process. The models are kept up-to-date using monitoring data, making manual error-prone parameter estimation unnecessary. The proposed approach offers a solid basis for implementing model-based autonomic performance and resource management techniques that continuously adapt the system during operation in order to ensure that performance objectives are met while efficiently using system resources. The vital benefit of employing models for system adaptation is that the performance models provide relevant information for what-if analyses and thus can drive the autonomic decision-making process. It is possible to search for valid and suitable system configurations on the model level and thus, unnecessary and possibly costly adaptations of the system can be avoided.

For example, the thesis of Huber (2014) is built directly on our approach, implementing a framework for autonomic performance-aware resource management. Huber (2014) evaluated the framework end-to-end in two different representative case studies (beyond the ones considered in this thesis), demonstrating that it can provide significant efficiency gains of up to 50% without sacrificing performance guarantees. Furthermore, it is shown that the approach enables proactive system adaptation, reducing the amount of SLA violations by 60% compared to a conventional trigger-based approach. The results of the case studies in Huber (2014) showed that it is possible to apply architecture-level performance models and online performance prediction to perform autonomic system adaptation on the model level such that the system's operational goals are maintained.

Other ongoing dissertation projects are going to use DML as a basis. Moreover, the work is already used in active collaborations with the industry applying it to real-life systems.

8.2 Open Questions and Future Work

The results of this thesis provide a basis for several areas of future work. In the following overview, we provide pointers for research extending our work.

Integration of Black-Box Performance Models

We presented existing work on black-box models in Section 2.1. There are many approaches in the literature that implement sophisticated extrapolation and interpolation techniques such as Classification and Regression Trees (CART), Multivariate Adaptive Regression Splines (MARS), Kriging models, or genetic programming. These techniques can be integrated to further improve the characterization of response times as part of our black-box behavior models or to improve the characterization of resource demands as part of our coarse-grained and fine-grained behavior models. The idea is to re-use existing work in order to provide further flexibility for parameter characterization. For instance, the application of MARS to characterize a response time distribution may allow further reducing the monitoring overhead, since the MARS method provides good approximations with only few data points.

Load-Dependent Resource Demands

In classical performance engineering, resource demands are typically assumed to be load-independent. However, modern processors implement Dynamic Voltage and Frequency Scaling (DVFS) mechanisms that adapt the processor speed depending on the current load. Thus, resource demands may appear to be load-dependent. To further increase the prediction accuracy, this load-dependency should be considered. Current versions of established model solvers such as SimQPN for QPNs or LQNS for LQNs are lacking support for solving performance models with load-dependent resource demands. Given that our performance prediction process builds on existing model solvers, this shortcoming is inherited. Hence, in order to support load-dependent resource demands, one should first extend the existing model solvers and then integrate the notion of a load-dependent resource demand in our model abstractions and resource demand estimation approaches.

Event-Based Systems

The work in Rathfelder (2013) describes how event-based interactions in component-based architectures can be modeled. It furthermore provides a generic approach how the developed modeling abstractions can be integrated into an architecture-level performance model. This approach can be applied to extend DML in order to add support for modeling event-based interactions such as point-to-point connections or decoupled publish/subscribe interactions. Platform-specific details about the event processing within the communication middleware are encapsulated.

Performance Data Repository

In Chapter 6, we described several techniques to integrate architecture-level performance models with system environments. However, an encompassing framework providing a performance data repository to store revisions of architecture-level performance models, run-time monitoring data, issued performance queries and corresponding prediction results would be valuable. Following the ideas of Woodside et al. (2007), the performance data repository shall aim at the convergence of performance monitoring, modeling and prediction as interrelated activities. With the techniques presented in Chapter 6 and Chapter 5, the prediction and model maintenance facilities are already provided as part of this thesis. Nevertheless, a performance data repository could improve the applicability and extendability of our work.

Automated Extraction of Performance Influences of Virtualization Platforms

As already discussed in Section 6.4.4, virtualization platforms present many challenges compared to native environments. There are many complex performance effects and influences in virtualized environments (e.g., mutual influences of the fine-granular system components and layers such as operation system, virtualization, middleware, application logic, I/O subsystem, caching and communication protocols). To improve the prediction accuracy in virtualized environments, such effects need to be better understood. Such effects are only observable during system operation when the system is running in the real production environment under real production workloads as opposed to running in a controlled testing environment with artificial workloads or synthetic benchmarks. Thus, the most promising approach to deal with the challenge of capturing the non-linear and multidimensional performance influences and interactions in the virtualization platform is to conduct the model (parameter) extraction process, possibly as part of the virtualization platform itself, at system run-time.

Integration of Specialized Resource Modeling Approaches

As part of ongoing research projects, suitable modeling abstractions for network infrastructures (Rygielski and Kounev, 2014) and storage systems (Noorshams et al., 2014) are under development. Given that these modeling approaches focus on network models respectively storage models, they aim to support: (i) more accurate performance analysis than what is possible with coarse-grained resource models, and (ii) further degrees-of-freedom when evaluating fine-granular configuration options of network infrastructures or storage systems. To obtain performance predictions, these specialized performance models require detailed workload profiles as input. Using DML, such workload profiles can be derived from the modeled application layer and the corresponding usage profile. These specialized modeling approaches should be integrated in DML, on the one hand, to increase the modeling capabilities of DML, on the other hand, to simplify the applicability of the specialized models.

QoS Properties Beyond Performance

The presented approach is focused on performance prediction, however, the general modeling approach developed in this thesis is not limited to performance. In future work, DML could be extended to support the analysis of further QoS properties. For instance, architecture-based reliability analysis (Brosch et al., 2011) could be integrated in DML in order to support evaluations of trade-offs between performance and reliability. For example, database transactions failed due to optimistic locking can be retried multiple times. This may increase reliability at the cost of performance. Other system properties such as power consumption and operating costs are gaining in importance. In particular, adding cost estimates to DML would allow multi-criteria optimizations trading-off between performance and costs (cf. Koziolek et al. (2013)).

Self-Aware Computing Systems

The long-term vision of the Descartes Research Project — the research project that funded this thesis — is to develop new methods for the engineering of self-aware computing systems. The latter are designed with built-in online QoS prediction and self-adaptation capabilities used to enforce QoS requirements in a cost- and energy-efficient manner. Self-awareness, in this context, is defined by the combination of the following three properties (Kounev, 2011):

- Self-reflective: i) aware of their software architecture, execution environment and the hardware infrastructure on which they are running, ii) aware of their operational goals in terms of QoS requirements, SLAs and cost- and energy-efficiency targets, iii) aware of dynamic changes in the above during operation,

- Self-predictive: able to predict the effect of dynamic changes (e.g., changing service workloads or QoS requirements) as well as predict the effect of possible adaptation actions (e.g., changing service deployment and/or resource allocations),
- Self-adaptive: proactively adapting as the environment evolves in order to ensure that their QoS requirements and respective SLAs are continuously satisfied while at the same time operating costs and energy-efficiency are optimized.

The concepts presented in this work as well as in the thesis of Huber (2014) lay the foundation for this vision. In the future, self-aware computing systems should be designed from the ground up with built-in self-reflective, self-predictive, and self-adaptive capabilities. Furthermore, the overall approach should be applied in industrial cooperations to showcase the applicability of our approach and thereby establish the vision of the self-aware computing paradigm.

List of Acronyms and Abbreviations

API	Application Programming Interface.
AQuA	Automatic Quality Assurance.
AWR	Automatic Workload Repository.
CBML	Component-Based Modeling Language.
CCL	Component Composition Language.
CGSPN	Colored GSPN.
CPN	Colored PN.
CRM	Customer Relationship Management.
DBS	Database Server.
DML	Descartes Modeling Language.
DNS	Domain Name System.
DoF	Degree-of-Freedom.
DQL	Descartes Query Language.
DVFS	Dynamic Voltage and Frequency Scaling.
EJB	Enterprise JavaBean.
ERP	Enterprise Resource Planning.
FCFS	First-Come-First-Served.
GSPN	Generalized Stochastic PN.
IS	Infinite-Server.
Java EE	Java Enterprise Edition.
JDBC	Java Database Connectivity.
JMS	Java Message Service.
JPA	Java Persistence API.
JSP	Java Server Pages.
JVM	Java Virtual Machine.
KLAPER	Kernel LAnguage for PErformance and Reliability analysis.
LAD	Least Absolute Differences.
LQN	Layered Queueing Network.
LSQ	Least Squares.
MARS	Multivariate Adaptive Regression Splines.
MDB	Message-Driven Bean.
MLE	Maximum Likelihood Estimation.
MVA	Mean Value Analysis.
OCL	Object Constraint Language.
PCM	Palladio Component Model.
PDF	Probability Density Function.
PECT	Prediction Enabled Component Technology.

PMF	Probability Mass Function.
PN	Petri Net.
PS	Processor-Sharing.
QEE	Query Execution Engine.
QN	Queueing Network.
QoS	Quality of Service.
QPME	Queueing Petri Net Modeling Environment.
QPN	Queueing Petri Net.
RDSEFF	Resource Demanding Service Effect Specification.
RMI	Remote Method Invocation.
SaaS	Software-as-a-Service.
SAN	Storage Area Network.
SLA	Service Level Agreement.
SOAP	Simple Object Access Protocol.
SPA	Stochastic Process Algebra.
SPE	Software Performance Engineering.
SPEC	Standard Performance Evaluation Corporation.
SPN	Stochastic Petri Net.
StoEx	Stochastic Expression.
SVM	Support Vector Machine.
UML	Unified Modeling Language.
UML-SPT	UML Profile for Schedulability, Performance and Time.
vCPU	virtual Central Processing Unit.
VM	Virtual Machine.
WLDF	WebLogic Diagnostics Framework.
WLS	Oracle WebLogic Server.

List of Figures

List of Tables

Bibliography

Abdelzaher, T., Shin, K., and Bhatti, N. (2002). Performance guarantees for web server end-systems: a control-theoretical approach. *Parallel and Distributed Systems, IEEE Transactions on*, 13(1):80–96.

Abrahao, B., Almeida, V., Almeida, J., Zhang, A., Beyer, D., and Safai, F. (2006). Self-adaptive sla-driven capacity management for internet services. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 557–568.

Adam, C. and Stadler, R. (2006). A Middleware Design for Large-scale Clusters Offering Multiple Services. *IEEE electronic Transactions on Network and Service Management*, 3(1).

Ali, R. A., Amin, K., von Laszewski, G., Rana, O., Walker, D., Hategan, M., and Zaluzec, N. (2004). Analysis and Provision of QoS for Distributed Grid Applications. *Journal of Grid Computing*, 2(2).

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7):1–19.

Almeida, J., lio Almeida, V., Ardagna, D., Cunha, Ã., Francalanci, C., and Trubian, M. (2010). Joint admission control and resource allocation in virtualized servers. *Journal of Parallel and Distributed Computing*, 70(4):344 – 362.

Amazon Web Services (2010). Amazon auto scaling. `http://aws.amazon.com/documentation/autoscaling/`. Last visit: 2014-03-22.

Anderson, E., Hoover, C., Li, X., and Tucek, J. (2009). Efficient tracing and performance analysis for large distributed systems. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–10.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

Balsamo, S., Di Marco, A., Inverardi, P., and Simeoni, M. (2004). Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5).

Balsamo, S. and Marzolla, M. (2003). A simulation-based approach to software performance modeling. *SIGSOFT Softw. Eng. Notes*, 28(5):363–366.

Balsamo, S., Marzolla, M., and Mirandola, R. (2006). Efficient performance models in component-based software engineering. In *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*, pages 64–71.

Barcelona Supercomputing Center (2014). Paraver: a flexible performance analysis tool. `http://www.bsc.es/computer-sciences/performance-tools/paraver/general-overview`. Last visit: 2014-04-10.

Bard, Y. and Shatzoff, M. (1978). Statistical Methods in Computer Performance Analysis. *Current Trends in Programming Methodology*, III.

Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. (2004). Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA. USENIX Association.

Barroso, L. A. and Hölzle, U. (2007). The case for energy-proportional computing. *Computer*, 40(12):33–37.

Bause, F. (1993). Queueing Petri Nets˜- A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models, Toulouse, France, October 19-22.*

Bause, F. and Kritzinger, F. (2002). *Stochastic Petri Nets˜- An Introduction to the Theory.* Vieweg Verlag, second edition.

Becker, M., Becker, S., and Meyer, J. (2013a). SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems. In *Proceedings of Software Engineering 2013 (SE2013), Aachen.*

Becker, M., Luckey, M., and Becker, S. (2013b). Performance analysis of self-adaptive systems for requirements validation at design-time. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pages 43–52, New York, NY, USA. ACM.

Becker, S., Grunske, L., Mirandola, R., and Overhage, S. (2004). Performance prediction of component-based systems - a survey from an engineering perspective. In *Architecting Systems with Trustworthy Components.*

Becker, S., Happe, J., and Koziolek, H. (2006). Putting Components into Context: Supporting QoS-Predictions with an explicit Context Model. In Reussner, R., Szyperski, C., and Weck, W., editors, *Proceedings 11th International Workshop on Component Oriented Programming (WCOP'06)*, pages 1–6.

Becker, S., Koziolek, H., and Reussner, R. (2007). Model-based performance prediction with the palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, pages 54–65, New York, NY, USA. ACM.

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

Bennani, M. N. and Menascé, D. (2004). Assessing the robustness of self-managing computer systems under highly variable workloads. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 62–69, Washington, DC, USA. IEEE Computer Society.

Bennani, M. N. and Menascé, D. (2005). Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 229–240, Washington, DC, USA. IEEE Computer Society.

Berbner, R., Spahn, M., Repp, N., Heckmann, O., and Steinmetz, R. (2006). Heuristics for QoS-aware Web Service Composition. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 72–82, Washington, DC, USA. IEEE Computer Society.

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

Bertolino, A. and Mirandola, R. (2004). CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Edinburgh, UK*, volume 3054 of *LNCS*, pages 233–248.

Blair, G., Bencomo, N., and France, R. (2009). Models@run.time. *Computer*, 42(10):22–27.

Bolch, G., Greiner, S., de Meer, H., and Trivedi, K. S. (1998). *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley-Interscience, New York, NY, USA.

Bondarev, E., Muskens, J., With, P. d., Chaudron, M., and Lukkien, J. (2004). Predicting real-time properties of component assemblies: A scenario-simulation approach. In *EUROMICRO*, pages 40–47.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Chapman & Hall, New York, NY.

Briand, L., Labiche, Y., and Leduc, J. (2006). Toward the reverse engineering of uml sequence diagrams for distributed java software. *Software Engineering, IEEE Transactions on*, 32(9):642–663.

Brooks, C. (2011). Cloud SLAs the next bugbear for enterprise IT. `http://searchcloudcomputing.techtarget.com/news/2240036361/Cloud-SLAs-the-next-bugbear-for-enterprise-IT`. Last visit: 2014-03-20.

Brosch, F., Koziolek, H., Buhnova, B., and Reussner, R. (2011). Architecture-based reliability prediction with the palladio component model. *Transactions on Software Engineering*, 38(6).

Brosig, F. (2009). Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. Master's thesis, Universität Karlsruhe (TH), Karlsruhe, Germany.

Brosig, F. (2011). Online performance prediction with architecture-level performance models. In Reussner, R., Pretschner, A., and Jähnichen, S., editors, *Software Engineering (Workshops) - Doctoral Symposium, February 21–25, 2011*, volume 184 of *Lecture Notes in Informatics (LNI)*, pages 279–284, Bonn, Germany. GI.

Brosig, F., Gorsler, F., Huber, N., and Kounev, S. (2013a). Evaluating Approaches for Performance Prediction in Virtualized Environments. In *Proceedings of the IEEE 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2013)*.

Brosig, F., Huber, N., and Kounev, S. (2011). Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE)*.

Brosig, F., Huber, N., and Kounev, S. (2012). Modeling Parameter and Context Dependencies in Online Architecture-Level Performance Models. In *Proceedings of the 15th ACM SIGSOFT International Symposium on Component Based Software Engineering (CBSE 2012), June 26–28, 2012, Bertinoro, Italy*.

Brosig, F., Huber, N., and Kounev, S. (2013b). Architecture-Level Software Performance Abstractions for Online Performance Prediction. *Elsevier Science of Computer Programming Journal (SciCo)*.

Brosig, F., Kounev, S., and Krogmann, K. (2009). Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *Proceedings of the*

1st International Workshop on Run-time mOdels for Self-managing Systems and Applications (ROSSA 2009). In conjunction with Fourth International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2009), Pisa, Italy, October 19, 2009. ACM, New York, NY, USA.

Brosig, F., Meier, P., Becker, S., Koziolek, A., Koziolek, H., and Kounev, S. (2014). Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures. *IEEE Transactions on Software Engineering.* Accepted for publication.

Bruno, N., Chaudhuri, S., and Gravano, L. (2001). Stholes: A multidimensional workload-aware histogram. *SIGMOD Rec.*, 30(2):211–222.

Bureau International des Poids et Mesures (2005). International vocabulary of metrology — Basic and general concepts and associated terms (VIM). `http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2008.pdf`. Last visit: 2014-02-17.

Carrera, D., Guitart, J., Torres, J., Ayguade, E., and Labarta, J. (2003). Complete instrumentation requirements for performance analysis of Web based technologies. In *Intl. Symp. on Perf. Anal. of Syst. and Softw.*

Casale, G., Cremonesi, P., and Turrin, R. (2007). How to Select Significant Workloads in Performance Models. In *CMG Conference Proceedings.*

Casale, G., Cremonesi, P., and Turrin, R. (2008). Robust Workload Estimation in Queueing Network Performance Models. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 183–187.

Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q., and Gautam, N. (2005). Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 303–314, New York, NY, USA. ACM.

Chouambe, L., Klatt, B., and Krogmann, K. (2008). Reverse engineering software-models of component-based systems. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 93–102.

Clark, A., Gilmore, S., Hillston, J., and Tribastone, M. (2007). Stochastic process algebras. In *Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation*, SFM'07, pages 132–179, Berlin, Heidelberg. Springer-Verlag.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Compuware (2008). Application Performance Management Survey. `http://www.docstoc.com/docs/425507/Application-Performance-Management-Survey-by-Compuware`. Last visit: 2014-05-12.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition.

Cortellessa, V. and Mirandola, R. (2000). Deriving a queueing network based performance model from uml diagrams. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 58–70, New York, NY, USA. ACM.

Courtois, M. and Woodside, M. (2000). Using regression splines for software performance analysis. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 105–114, New York, NY, USA. ACM.

Diaconescu, A. and Murphy, J. (2005). Automating the Performance Management of Component-Based Enterprise Systems through the use of Redundancy. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 44–53, New York, NY, USA. ACM Press.

Dilley, J., Friedrich, R., Jin, T., and Rolia, J. (1997). Measurement tools and modeling techniques for evaluating web server performance. In Marie, R., Plateau, B., Calzarossa, M., and Rubino, G., editors, *Computer Performance Evaluation Modelling Techniques and Tools*, volume 1245 of *Lecture Notes in Computer Science*, pages 155–168. Springer Berlin Heidelberg.

Ehlers, J. and Hasselbring, W. (2011). Self-adaptive software performance monitoring. In *Software Engineering, GI.*

Elkhodary, A., Esfahani, N., and Malek, S. (2010). Fusion: A framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 7–16, New York, NY, USA. ACM.

Eskenazi, E., Fioukov, A., and Hammer, D. (2004). Performance Prediction for Component Compositions. In *Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE7).*

Foster, I. and Kesselman, C. (2003). *The Grid 2: Blueprint for a New Computing Infrastructure.* The Elsevier Series in Grid Computing. Elsevier Science.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Franks, G. (1999). *Performance Analysis of Distributed Server Systems.* PhD thesis, Department of Systems and Computer Engineering, Carleton University,Ottawa, Ontario, Canada.

Franks, G., Majumdar, S., Neilson, Petriu, D., Rolia, J., and Woodside, M. (1996). Performance analysis of distributed server systems. In *In Proceedings of the 6th International Conference on Software Quality*, pages 15–26.

Franks, G., Maly, P., Woodside, M., Petriu, D. C., Hubbard, A., and Mroz, M. (2011). Layered Queueing Network Solver (LQNS) software package. `http://www.sce.carleton.ca/rads/lqns/`. Last visit: 2014-03-20.

Franks, G., Omari, T., Woodside, C. M., Das, O., and Derisavi, S. (2009). Enhanced modeling and solution of layered queueing networks. *IEEE Trans. on Software Engineering*, 35(2):148–161.

Freedman, D. and Diaconis, P. (1981). On the histogram as a density estimator:l 2 theory. *Zeitschrift fuer Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 57(4):453–476.

Gambi, A., Toffetti, G., Pautasso, C., and Pezze, M. (2013). Kriging controllers for cloud applications. *Internet Computing, IEEE*, 17(4):40–47.

Gibbons, P. B., Matias, Y., and Poosala, V. (2002). Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.*, 27(3):261–298.

Gilly, K., Alcaraz, S., Juiz, C., and Puigjaner, R. (2009). Analysis of burstiness monitoring and detection in an adaptive web system. *Comput. Netw.*, 53(5):668–679.

Gilmore, S., Haenel, V., Kloul, L., and Maidl, M. (2005). Choreographing Security and Performance Analysis for Web Services. In *EPEW and WS-FM*, LNCS.

Glass, R. L. (1998). *Software Runaways. Lessons learned from Massive Software Project Failures.* Prentice Hall.

Gorsler, F. (2013). Online Performance Queries for Architecture-Level Performance Models. Master's thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany.

Gorsler, F., Brosig, F., and Kounev, S. (2013). Controlling the palladio bench using the descartes query language. In Becker, S., Hasselbring, W., van Hoorn, A., and Reussner, R., editors, *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDAYS 2013)*, number 1083 in CEUR Workshop Proceedings, pages 109–118, Aachen, Germany. CEUR-WS.org.

Gorsler, F., Brosig, F., and Kounev, S. (2014). Performance queries for architecture-level performance models. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, New York, NY, USA. ACM.

Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA. ACM.

Grassi, V., Mirandola, R., and Sabetta, A. (2007). 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.

Gunopulos, D., Kollios, G., Tsotras, V. J., and Domeniconi, C. (2000). Approximating multi-dimensional aggregate range queries over real attributes. *SIGMOD Rec.*, 29(2):463–474.

Gunter, D. and Tierney, B. (2003). Netlogger: a toolkit for distributed system performance tuning and debugging. In *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*, pages 97–100.

Gupta, D., Gardner, R., and Cherkasova, L. (2005). XenMon: QoS monitoring and performance profiling tool. Technical Report HPL-2005-187, HP Labs.

Hall, R. J. (1992). Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 296–306, New York, NY, USA. ACM.

Hamlet, D. (2009). Tools and experiments supporting a testing-based theory of component composition. *ACM Trans. Softw. Eng. Methodol.*, 18(3):12:1–12:41.

Happe, J., Groenda, H., Hauck, M., and Reussner, R. (2010). A prediction model for software performance in symmetric multiprocessing environments. In *2010 Seventh International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 59–68.

Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

Herbst, N. R., Huber, N., Kounev, S., and Amrehn, E. (2013). Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*, pages 187–198, New York, NY, USA. ACM.

Herbst, N. R., Huber, N., Kounev, S., and Amrehn, E. (2014). Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. *Concurrency and Computation - Practice and Experience, Special Issue with extended versions of the best papers from ICPE 2013, John Wiley and Sons, Ltd.*

Herzog, U. (1990). Formal description, time and performance analysis a framework. In Härder, T., Wedekind, H., and Zimmermann, G., editors, *Entwurf und Betrieb verteilter Systeme*, volume 264 of *Informatik-Fachberichte*, pages 172–190. Springer Berlin Heidelberg.

Hillston, J. (1996). *A Compositional Approach to Performance Modelling.* Cambridge University Press, New York, NY, USA.

Hissam, S. A., Moreno, G. A., Stafford, J., and Wallnau, K. C. (2001). Packaging predictable assembly with prediction-enabled component technology. Technical report, Carnegie Mellon University.

Hissam, S. A., Moreno, G. A., Stafford, J. A., and Wallnau, K. C. (2002). Packaging Predictable Assembly. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 108–124, London, UK. Springer-Verlag.

Hollingsworth, J. K. and Miller, B. (1996). An adaptive cost system for parallel program instrumentation. In Bouge, L., Fraigniaud, P., Mignotte, A., and Robert, Y., editors, *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 88–97. Springer Berlin Heidelberg.

Hoorn, A. V., Rohr, M., and Hasselbring, W. (2008). 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 (SIPEW '08), volume 5119 of Lecture Notes in Computer Science (LNCS*, pages 124–143. SPEC, Springer. ISBN.

Hrischuk, C., Murray Woodside, C., and Rolia, J. (1999). Trace-based load characterization for generating performance software models. *Software Engineering, IEEE Transactions on*, 25(1):122–135.

Huber, N. (2014). *Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures.* PhD thesis, Karlsruhe Institute of Technology (KIT). To be published.

Huber, N., Brosig, F., and Kounev, S. (2011a). Model-based Self-Adaptive Resource Allocation in Virtualized Environments. In *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011)*, pages 90–99, New York, NY, USA. ACM.

Huber, N., Brosig, F., and Kounev, S. (2012a). Modeling Dynamic Virtualized Resource Landscapes. In *Proceedings of the 8th ACM SIGSOFT International Conference on the Quality of Software Architectures (QoSA 2012)*, pages 81–90, New York, NY, USA. ACM.

Huber, N., van Hoorn, A., Koziolek, A., Brosig, F., and Kounev, S. (2012b). S/T/A: Meta-Modeling Run-Time Adaptation in Component-Based System Architectures. In *Proceedings of the 9th IEEE International Conference on e-Business Engineering (ICEBE 2012)*, pages 70–77, Los Alamitos, CA, USA. IEEE Computer Society.

Huber, N., van Hoorn, A., Koziolek, A., Brosig, F., and Kounev, S. (2013). Modeling Run-Time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments. *Service Oriented Computing and Applications Journal (SOCA).*

Huber, N., von Quast, M., Brosig, F., and Kounev, S. (2010). Analysis of the Performance-Influencing Factors of Virtualization Platforms. In *The 12th International Symposium on Distributed Objects, Middleware, and Applications (DOA 2010)*, Crete, Greece. Springer Verlag.

Huber, N., von Quast, M., Hauck, M., and Kounev, S. (2011b). Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In *International Conference on Cloud Computing and Service Science*, CLOSER'11.

Hunt, C. and John, B. (2011). *Java Performance.* Java Series. Pearson Education.

Hyperic (2014). Hyperic. `http://www.hyperic.com`. Last visit: 2014-04-14.

Intel (2013). Intel VTune Amplifier XE 2013. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe`. Last visit: 2014-04-10.

Israr, T., Woodside, M., and Franks, G. (2007). Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(4):474–492.

Izenman, A. (2009). *Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning.* Springer Texts in Statistics. Springer.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley Professional Computing. Wiley.

Jamshidi, P., Ahmad, A., and Pahl, C. (2014). Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 95–104, New York, NY, USA. ACM.

Jordan, M. and Jacobs, R. A. (1993). Hierarchical mixtures of experts and the em algorithm. In *Neural Networks, 1993. IJCNN '93-Nagoya. Proceedings of 1993 International Joint Conference on*, volume 2, pages 1339–1344 vol.2.

Jung, G., Hiltunen, M., Joshi, K., Schlichting, R., and Pu, C. (2010). Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62 –73.

Jung, G., Joshi, K., Hiltunen, M., Schlichting, R., and Pu, C. (2008). Generating adaptation policies for multi-tier applications in consolidated server environments. In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 23–32.

Kalbasi, A., Krishnamurthy, D., Rolia, J., and Richter, M. (2011). MODE: Mix Driven Online Resource Demand Estimation. In *Proceedings of the 7th International Conference on Network and Services Management*, pages 1–9.

Kappler, T., Koziolek, H., Krogmann, K., and Reussner, R. H. (2008). Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In *Software Engineering 2008*, volume 121 of *Lecture Notes in Informatics*, pages 140–154, Munich, Germany. Bonner Köllen Verlag.

Kelly, T. and Zhang, A. (2006). Predicting performance in distributed enterprise applications. Technical report, HP Labs Tech Report.

Kendall, D. G. (1953). Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, 24(3):338–354.

Kephart, J., Chan, H., Das, R., Levine, D., Tesauro, G., Rawson, F., and Lefurgy, C. (2007). Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, pages 24–24.

Kounev, S. (2005). *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction*. Shaker Verlag, Ph.D. Thesis, Technische Universität Darmstadt, Germany.

Kounev, S. (2006). Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502.

Kounev, S. (2011). Engineering of Self-Aware IT Systems and Services: State-of-the-Art and Research Challenges. In *Proceedings of the 8th European Performance Engineering Workshop (EPEW'11), Borrowdale, The English Lake District, October 12–13*. (Keynote Talk).

Kounev, S., Bender, K., Brosig, F., Huber, N., and Okamoto, R. (2011). Automated simulation-based capacity planning for enterprise data fabrics. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 27–36, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Kounev, S., Brosig, F., and Huber, N. (2014). Descartes Modeling Language (DML). Technical report, Karlsruhe Institute of Technology (KIT). To be published.

Kounev, S., Brosig, F., Huber, N., and Reussner, R. (2010). Towards self-aware performance and resource management in modern service-oriented systems. In *Proceedings of the 7th IEEE International Conference on Services Computing (SCC 2010), July 5-10, Miami, Florida, USA*. IEEE Computer Society.

Kounev, S. and Buchmann, A. (2003). 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 ˜- ISPASS2003, Austin, Texas, USA, March 20-22*.

Kounev, S. and Buchmann, A. (2006). SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364–394.

Kounev, S., Nou, R., and Torres, J. (2007). Autonomic QoS-Aware Resource Management in Grid Computing using Online Performance Models. In *2nd International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2007), October 23th-25th, Nantes, France, ISBN: 978-1-59593-819-0*.

Kounev, S., Sachs, K., Bacon, J., and Buchmann, A. (2008). A Methodology for Performance Modeling of Distributed Event-Based Systems. In *11th IEEE Intl. Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 13–22.

Koziolek, A., Ardagna, D., and Mirandola, R. (2013). Hybrid multi-attribute QoS optimization in component based software systems. *Journal of Systems and Software*, 86(10).

Koziolek, A. and Reussner, R. (2011). Towards a generic quality optimisation framework for component-based system models. In Crnkovic, I., Stafford, J. A., Bertolino, A., and Cooper, K. M. L., editors, *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, CBSE '11, pages 103–108, New York, NY, USA. ACM, New York, NY, USA.

Koziolek, H. (2008). *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, Germany.

Koziolek, H. (2010). Performance Evaluation of Component-based Software Systems: A Survey. *Performance Evaluation*, 67(8):634–658.

Koziolek, H., Happe, J., and Becker, S. (2006). Parameter dependent performance specifications of software components. In Hofmeister, C., Crnkovic, I., and Reussner, R., editors, *Quality of Software Architectures*, volume 4214 of *Lecture Notes in Computer Science*, pages 163–179. Springer Berlin Heidelberg.

Kraft, S., Pacheco-Sanchez, S., Casale, G., and Dawson, S. (2009). Estimating service resource consumption from response time measurements. In *VALUETOOLS '09: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 1–10.

Krebs, R., Momm, C., and Kounev, S. (2012a). Architectural Concerns in Multi-Tenant SaaS Applications. In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*. SciTePress.

Krebs, R., Momm, C., and Kounev, S. (2012b). Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments. In Buhnova, B. and Vallecillo, A., editors, *Proceedings of the 8th ACM SIGSOFT International Conference on the Quality of Software Architectures (QoSA 2012)*, pages 91–100, New York, USA. ACM Press.

Krogmann, K., Kuperberg, M., and Reussner, R. (2008). Reverse Engineering of Parametric Behavioural Service Performance Models from Black-Box Components. In Steffens, U., Addicks, J. S., and Streekmann, N., editors, *MDD, SOA und IT-Management (MSI 2008)*, pages 57–71, Oldenburg. GITO Verlag.

Krogmann, K., Kuperberg, M., and Reussner, R. (2010). Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Trans. Softw. Eng.*, 36(6):865–877.

Kumar, D., Tantawi, A., and Zhang, L. (2009a). Real-time performance modeling for adaptive software systems. In *VALUETOOLS '09: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 1–10.

Kumar, D., Zhang, L., and Tantawi, A. (2009b). Enhanced inferencing: estimation of a workload dependent performance model. In *VALUETOOLS '09: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 1–10.

Kuperberg, M., Krogmann, K., and Reussner, R. (2008a). Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE 2008), Karlsruhe, Germany, 14th-17th October 2008*, volume 5282 of *LNCS*, pages 48–63. Springer, Heidelberg.

Kuperberg, M., Krogmann, M., and Reussner, R. (2008b). ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)*.

Kyte, T. (2005). *Expert Oracle Database Architecture*. Expert's voice in Oracle. Apress.

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Li, J., Chinneck, J., Woodside, M., Litoiu, M., and Iszlai, G. (2009). Performance model driven QoS guarantees and optimization in clouds. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 15–22, Washington, DC, USA. IEEE Computer Society.

Lientz, B. P. and Swanson, E. B. (1981). Problems in application software maintenance. *Commun. ACM*, 24(11):763–769.

Little, J. D. C. (1961). A proof for the queuing formula: L= λ W. *Operations Research*, 9(3):pp. 383–387.

Liu, H. H. (2009). *Software Performance and Scalability: A Quantitative Approach.* Wiley Publishing.

Liu, Y., Fekete, A., and Gorton, I. (2005). Design-level performance prediction of component-based applications. *IEEE Trans. Softw. Eng.*, 31(11):928–941.

Liu, Z., Wynter, L., Xia, C. H., and Zhang, F. (2006). Parameter inference of queueing models for IT systems using end-to-end measurements. *Performance Evaluation*, 63(1):36–60.

Liu, Z., Xia, C. H., Momcilovic, P., and Zhang, L. (2003). AMBIENCE: Automatic Model Building using IferENCE. Technical report, IBM Research.

López-Grao, J. P., Merseguer, J., and Campos, J. (2004). From uml activity diagrams to stochastic petri nets: Application to software performance engineering. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, pages 25–36, New York, NY, USA. ACM.

Lu, L., Zhang, H., Jiang, G., Chen, H., Yoshihira, K., and Smirni, E. (2011). Untangling mixed information to calibrate resource utilization in virtual machines. In *Int. Conf. on Autonomic Computing.*

Malony, A. D., Shende, S., Morris, A., and Wolf, F. (2007). Compensation of measurement overhead in parallel performance profiling. *International Journal of High Performance Computing Applications*, 21(2):174–194.

Meier, P. (2010). Automated Transformation of Palladio Component Models to Queueing Petri Nets. Master's thesis, Karlsruhe Institute of Technology (KIT).

Meier, P., Kounev, S., and Koziolek, H. (2011). Automated Transformation of Palladio Component Models to Queueing Petri Nets. In *In 19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011).*

Menascé, D. (2008). Computing missing service demand parameters for performance models. In *CMG Conference Proceedings*, pages 241–248.

Menascé, D. and Bennani, M. (2003). On the use of performance models to design self-managing computer systems. In *Proceedings of the Computer Measurement Group Conference (CMG), Dallas, Texas*, pages 7–12.

Menascé, D., Bennani, M. N., and Ruan, H. (2005). *Self-Star Properties in Complex Information Systems*, volume 3460 of *LNCS*, chapter On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems. Springer Verlag.

Menascé, D., Ruan, H., and Gomaa, H. (2004a). A framework for QoS-aware software components. *SIGSOFT Softw. Eng. Notes*, 29(1):186–196.

Menascé, D., Ruan, H., and Gomaa, H. (2007). Qos management in service-oriented architectures. *Performance Evaluation*, 64(7-8):646–663.

Menascé, D. A., Almeida, V. A. F., and Dowdy, L. W. (1994). *Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems.* Prentice Hall, Englewood Cliffs, NG.

Menascé, D. A., Almeida, V. A. F., and Dowdy, L. W. (2004b). *Performance by Design.* Prentice Hall.

Menasce, D. A. and Virgilio, A. F. A. (2000). *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Mi, H., Wang, H., Yin, G., Zhou, Y., Shi, D., and Yuan, L. (2010). Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 514–521.

Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. (1995). The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46.

Milner, R. (1989). *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall International.

Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2012). *Foundations of Machine Learning.* The MIT Press.

Mos, A. (2004). *A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications.* PhD thesis, Dublin City University, Ireland.

Mos, A. and Murphy, J. (2002a). A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 235–236, New York, NY, USA. ACM Press.

Mos, A. and Murphy, J. (2002b). Understanding performance issues in component-oriented distributed applications: The compas framework. In *Seventh International Workshop on Component-Oriented Programming (WCOP) of the 16th European Conference on Object-Oriented Programming (ECOOP), Malaga, Spain.*

Murugesan, S. (2008). Harnessing Green IT: Principles and Practices. *IT Professional*, vol. 10, no. 1:24–33.

Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. (2007). Understanding measurement perturbation in trace-based data. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6.

Nadeem, F., Yousaf, M., Prodan, R., and Fahringer, T. (2006). Soft benchmarks-based application performance prediction using a minimum training set. In *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on*, pages 71–71.

Noorshams, Q., Reeb, R., Rentschler, A., Kounev, S., and Reussner, R. (2014). Enriching Software Architecture Models with Statistical Models for Performance Prediction in Modern Storage Environments. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*, CBSE '14. (Paper accepted for publication).

Nou, R., Kounev, S., Julia, F., and Torres, J. (2009). Autonomic QoS control in enterprise Grid environments using online simulation. *Journal of Systems and Software*, 82(3):486–502.

Object Management Group (OMG) (2005). UML-SPT: UML Profile for Schedulability, Performance, and Time, v1.1.

Object Management Group (OMG) (2006). UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE).

Othman, A., Dew, P., Djemamem, K., and Gourlay, I. (2003). Adaptive Grid Resource Brokering. In *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, pages 172–179.

Pacifici, G., Segmuller, W., Spreitzer, M., and Tantawi, A. N. (2008). Cpu demand for web serving: Measurement analysis and dynamic estimation. *Performance Evaluation.*

Pacifici, G., Spreitzer, M., Tantawi, A., and Youssef, A. (2005). Performance Management of Cluster-Based Web Services. *IEEE Journal on Selected Areas in Communications*, 23(12):2333–2343.

Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45.

Perez, J. F., Pacheco-Sanchez, S., and Casale, G. (2013). An offline demand estimation method for multi-threaded applications. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS).*

Petri, C. (1962). *Kommunikation mit Automaten.* Schriften des Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn. Rhein.-Westfäl. Inst. f. Instrumentelle Mathematik an der Univ. Bonn.

Poosala, V., Haas, P. J., Ioannidis, Y. E., and Shekita, E. J. (1996). Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 25(2):294–305.

Rathfelder, C. (2013). *Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation*, volume 10 of *The Karlsruhe Series on Software Design and Quality.* KIT Scientific Publishing, Karlsruhe, Germany.

Reussner, R., Becker, S., Burger, E., Happe, J., Hauck, M., Koziolek, A., Koziolek, H., Krogmann, K., and Kuperberg, M. (2011). The Palladio Component Model. Technical report, KIT, Fakultät für Informatik, Karlsruhe.

Reussner, R. H. (2001). The use of parameterised contracts for architecting systems with software components. In *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01.*

Rohr, M., Hoorn, A., Giesecke, S., Matevska, J., Hasselbring, W., and Alekseev, S. (2008). Trace-context sensitive performance profiling for enterprise software applications. In Kounev, S., Gorton, I., and Sachs, K., editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 283–302. Springer Berlin Heidelberg.

Rolia, J., Kalbasi, A., Krishnamurthy, D., and Dawson, S. (2010a). Resource demand modeling for multi-tier services. In *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 207–216. ACM.

Rolia, J., Krishnamurthy, D., Casale, G., and Dawson, S. (2010b). BAP: a benchmark-driven algebraic method for the performance engineering of customized services. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 3–14. ACM.

Rolia, J. and Sevcik, K. (1995). The method of layers. *Software Engineering, IEEE Transactions on*, 21(8):689–700.

Rolia, J. and Vetland, V. (1995). Parameter estimation for performance models of distributed application systems. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '95. IBM Press.

Rolia, J. and Vetland, V. (1998). Correlating resource demand information with ARM data for application services. In *Proceedings of the 1st international workshop on Software and performance*, pages 219–230. ACM.

Rometsch, F. and Sauer, H. (2008). Dynatrace Diagnostics: Performance-Management und Fehlerdiagnose vereint. *iX*, 9/2008.

Rygielski, P. and Kounev, S. (2014). Data Center Network Throughput Analysis using Queueing Petri Nets. In *34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Wokrshops). 4th International Workshop on Data Center Performance, (DCPerf 2014)*. (Paper accepted for publication).

Schmeiser, B. (1982). Batch size effects in the analysis of simulation output. *Operations Research*, 30(3):pp. 556–568.

Schroeder, B., Wierman, A., and Harchol-Balter, M. (2006). Open versus closed: a cautionary tale. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, Berkeley, CA, USA. USENIX Association.

Scott, D. W. (1979). On optimal and data-based histograms. *Biometrika*, 66(3):605–610.

Sheikh, F. and Woodside, M. (1997). Layered analytic performance modelling of a distributed database system. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 482–490.

Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311.

Shivam, P., Babu, S., and Chase, J. S. (2006). Learning application models for utility resource planning. In *ICAC '06: Proceedings of the IEEE International Conference on Autonomic Computing*, pages 255–264.

Shousha, C., Petriu, D., Jalnapurkar, A., and Ngo, K. (1998). Applying performance modelling to a telecommunication system. In *Proceedings of the 1st International Workshop on Software and Performance*, WOSP '98, pages 1–6, New York, NY, USA. ACM.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2008). *Operating System Concepts*. Wiley Publishing, 8th edition.

Simon, D. (2006). *Optimal state estimation : Kalman, H. [infinity] and nonlinear approaches*. Wiley-Interscience, Hoboken, NJ.

Sitaraman, M., Kulczycki, G., Krone, J., Ogden, W. F., and Reddy, A. L. N. (2001). Performance Specification of Software Components. *SIGSOFT Softw. Eng. Notes*, 26(3):3–10.

Sitaraman, M. and Weide, B. (1994). Component-based software using resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22.

Smith, C. U. (2002). *Encyclopedia of Software Engineering*, chapter Software Performance Engineering, pages 1545–1562. John Wiley & Sons, 2nd edition.

Smith, C. U. and Williams, L. G. (1997). Performance engineering evaluation of object-oriented systems with spe*ed. In *Computer Performance Evaluation*, pages 135–154.

Smith, C. U. and Williams, L. G. (2002). *Performance Solutions˜- A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley.

Smola, A. J. and Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222.

Song, H. G., Ryu, Y., Chung, T. S., Jou, W., and Lee, K. (2005). Metrics, Methodology, and Tool for Performance-Considered Web Service Composition. In *Proceedings of the 20th International Symposium on Computer and Information Sciences (ISCIS 2005), Istanbul, Turkey, October 26-28*, volume 3733 of *LNCS*, pages 392–401. Springer.

SPEC (2010). SPECjEnterprise2010 Design Document. `http://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html`. Version 2010-05-20. Last visit: 2014-03-07.

SPEC (2014). Standard Performance Evaluation Corporation (SPEC). `http://www.spec.org/`. Version 2014-02-14. Last visit: 2014-03-07.

Spinner, S. (2011). Evaluating Approaches to Resource Demand Estimation. Master's thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany.

Spinner, S., Kounev, S., and Meier, P. (2012). Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0. In Haddad, S. and Pomello, L., editors, *Proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2012)*, volume 7347 of *Lecture Notes in Computer Science (LNCS)*, pages 388–397, Berlin, Heidelberg. Springer-Verlag.

Sriganesh, R., Brose, G., and Silverman, M. (2006). *Mastering Enterprise JavaBeans 3.0*. John Wiley & Sons.

Stewart, C., Kelly, T., and Zhang, A. (2007). Exploiting nonstationarity for performance prediction. *SIGOPS Oper. Syst. Rev.*, 41(3):31–44.

Sturges, H. A. (1929). The Choice of a Class Interval. *Journal of the American Statistical Association*, 21(153).

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA. IEEE Computer Society Press.

Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2 edition.

Tesauro, G., Jong, N. K., Das, R., and Bennani, M. N. (2006). A hybrid reinforcement learning approach to autonomic resource allocation. In *ICAC '06: IEEE International Conference on Autonomic Computing*, pages 65–73.

Thereska, E., Narayanan, D., and Ganger, G. (2005). Towards self-predicting systems: What if you could ask "what-if"? In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pages 196–200.

Tonella, P., Torchiano, M., Du Bois, B., and Systä, T. (2007). Empirical studies in reverse engineering: State of the art and future trends. *Empirical Software Engineering*, 12(5):551–571.

Trivedi, K. S. (2002). *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, Inc., second edition.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2007). Analytic modeling of multitier internet applications. *ACM Trans. Web*, 1(1).

van Hoorn, A. (2014a). *Model-Driven Online Capacity Management for Component-Based Software Systems*. PhD thesis, Faculty of Engineering, Kiel University. To be published.

van Hoorn, A. (2014b). *Online Capacity Management for Increased Resource Efficiency of Component-Based Software Systems*. PhD thesis, University of Kiel, Germany.

van Hoorn, A., Rohr, M., Gul, A., and Hasselbring, W. (2009). An adaptation framework enabling resource-efficient operation of software systems. In *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, WUP '09, pages 41–44, New York, NY, USA. ACM.

Verma, A., Ahuja, P., and Neogi, A. (2008). pmapper: Power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 243–264, New York, NY, USA. Springer-Verlag New York, Inc.

VMware (2006). Resource Management with VMware DRS. `http://www.vmware.com/pdf/vmware_drs_wp.pdf`. Version 2006-06-05. Last visit: 2014-04-30.

von Massow, R., van Hoorn, A., and Hasselbring, W. (2011). Performance simulation of runtime reconfigurable component-based software architectures. In *Software Architecture: Proceedings of the 5th European Conference on Software Architecture (ECSA 2011)*, pages 43–58. Springer Berlin/Heidelberg.

Wallnau, K. C. and Ivers, J. (2003). Snapshot of ccl: A language for predictable assembly. Technical Note CMU/SEI-2003-TN-025, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Wang, W., Huang, X., Qin, X., Zhang, W., Wei, J., and Zhong, H. (2012). Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, pages 439 –446.

Westermann, D. (2013). *Deriving Goal-oriented Performance Models by Systematic Experimentation*. PhD thesis, Karlsruhe Institute of Technology (KIT).

Westermann, D., Happe, J., Krebs, R., and Farahbod, R. (2012). Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 190–199, New York, NY, USA. ACM.

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B., and Bruel, J. (2009). Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. 17th IEEE International*, pages 79–88.

Wood, T., Cherkasova, L., Ozonat, K., and Shenoy, P. (2008). Profiling and modeling resource usage of virtualized applications. In Issarny, V. and Schantz, R., editors, *Middleware 2008*, volume 5346 of *Lecture Notes in Computer Science*, pages 366–387. Springer Berlin Heidelberg.

Woodside, C., Neilson, J., Petriu, D., and Majumdar, S. (1995). The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *Computers, IEEE Transactions on*, 44(1):20–34.

Woodside, M., Franks, G., and Petriu, D. (2007). The Future of Software Performance Engineering. In *Future of Software Engineering (FOSE'07)*, pages 171–187, Los Alamitos, CA, USA. IEEE Computer Society.

Woodside, M., Zheng, T., and Litoiu, M. (2006). Service System Resource Management Based on a Tracked Layered Performance Model. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 175–184.

Wu, X. and Woodside, M. (2004). Performance Modeling from Software Components. In *WOSP '04: Proceedings of the fourth International Workshop on Software and Performance*, volume 29, pages 290–301, New York, NY, USA. ACM Press.

Wynter, L., Xia, C. H., and Zhang, F. (2004). Parameter inference of queueing models for IT systems using end-to-end measurements. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 408–409.

Xu, J., Oufimtsev, A., Woodside, M., and Murphy, L. (2005). Performance modeling and prediction of enterprise javabeans with layered queuing network templates. *SIGSOFT Softw. Eng. Notes*, 31(2).

Zenoss (2014). Zenoss Software. `http://www.zenoss.com`. Last visit: 2014-04-14.

Zhang, L., Xia, C. H., Squillante, M. S., and Iii, W. N. M. (2002). Workload Service Requirements Analysis: A Queueing Network Optimization Approach. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, page 23ff.

Zhang, L., Zhang, B., Pahl, C., Xu, L., and Zhu, Z. (2013). Personalized quality prediction for dynamic service management based on invocation patterns. In Basu, S., Pautasso, C., Zhang, L., and Fu, X., editors, *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin Heidelberg.

Zhang, Q., Cherkasova, L., and Smirni, E. (2007). A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Proceedings of the Fourth International Conference on Autonomic Computing*, page 27ff.

Zheng, T., Woodside, C., and Litoiu, M. (2008). Performance Model Estimation and Tracking Using Optimal Filters. *Software Engineering, IEEE Transactions on*, 34(3):391–406.

Zheng, T., Yang, J., Woodside, M., Litoiu, M., and Iszlai, G. (2005). Tracking time-varying parameters in software systems with extended Kalman filters. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 334–345. IBM Press.