Karlsruhe Institute of Technology

Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften/
Doktors der Naturwissenschaften

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

Dissertation

von

Nikolaus Matthias Huber
aus Zell am Harmersbach

Tag der mündlichen Prüfung:	16.07.2014
Erster Gutachter:	Prof. Dr. Samuel Kounev
Zweiter Gutachter:	Prof. Dr. Ralf Reussner

Ich versichere wahrheitsgemäß, die Dissertation bis auf die dort angegebenen Hilfen selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 30. September 2014

Nikolaus Huber

Contents

Publication List

[Huber et al., 2014] Huber, N., van Hoorn, A., Koziolek, A., Brosig, F., and Kounev, S. (2014). Modeling Run-Time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments. *Service Oriented Computing and Applications Journal (SOCA)*, 8(1):73–89.

[Huber et al., 2012d] Huber, N., von Quast, M., Brosig, F., Hauck, M., and Kounev, S. (2012d). A Method for Experimental Analysis and Modeling of Virtualization Performance Overhead. In Ivanov, I., van Sinderen, M., and Shishkov, B., editors, *Cloud Computing and Services Science*, Service Science: Research and Innovations in the Service Economy, pages 353–370. Springer, New York.

[Huber et al., 2012c] Huber, N., van Hoorn, A., Koziolek, A., Brosig, F., and Kounev, S. (2012c). 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. Acceptance Rate (Full Paper): 19.7% (26/132).

[Huber et al., 2012b] Huber, N., Brosig, F., and Kounev, S. (2012b). 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. Acceptance Rate (Full Paper): 25.6%.

[Huber et al., 2012a] Huber, N., Brosig, F., Dingle, N., Joshi, K., and Kounev, S. (2012a). Providing Dependability and Performance in the Cloud: Case Studies. In Wolter, K., Avritzer, A., Vieira, M., and van Moorsel, A., editors, *Resilience Assessment and Evaluation of Computing Systems*, XVIII. Springer-Verlag, Berlin, Heidelberg. ISBN: 978-3-642-29031-2.

[Huber et al., 2011b] Huber, N., von Quast, M., Hauck, M., and Kounev, S. (2011b). Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER 2011)*, pages 563 – 573. SciTePress. Acceptance Rate: 18/164 = 10.9%, Best Paper Award.

[Huber et al., 2011a] Huber, N., Brosig, F., and Kounev, S. (2011a). Model-based Self-Adaptive Resource Allocation in Virtualized Environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011)*, pages 90–99, New York, NY, USA. ACM. Acceptance Rate (Full Paper): 27% (21/76).

[Huber et al., 2010b] Huber, N., von Quast, M., Brosig, F., and Kounev, S. (2010b). Analysis of the Performance-Influencing Factors of Virtualization Platforms. In *Proceedings of the 12th International Symposium on Distributed Objects, Middleware, and Applications (DOA 2010)*, Crete, Greece. Springer Verlag. Acceptance Rate (Full Paper): 33%.

[Huber et al., 2010a] Huber, N., Becker, S., Rathfelder, C., Schweflinghaus, J., and Reussner, R. (2010a). Performance Modeling in Industry: A Case Study on Storage Virtualization. In *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010), Software Engineering in Practice Track*, pages 1–10, New York, NY, USA. ACM. Acceptance Rate (Full Paper): 23% (16/71).

[Huber, 2009] Huber, N. (2009). Performance Modeling of Storage Virtualization. Master's thesis, Universität Karlsruhe (TH), Karlsruhe, Germany. GFFT Prize.

[Brosig et al., 2013a] 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).* (Short Paper).

[Brosig et al., 2011] Brosig, F., Huber, N., and Kounev, S. (2011). Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, Oread, Lawrence, Kansas. Acceptance Rate (Full Paper): 14.7% (37/252).

[Brosig et al., 2012] 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.* Acceptance Rate (Full Paper): 28.5%.

[Brosig et al., 2013b] Brosig, F., Huber, N., and Kounev, S. (2013b). Architecture-Level Software Performance Abstractions for Online Performance Prediction. *Elsevier Science of Computer Programming Journal (SciCo).*

[Hauck et al., 2011] Hauck, M., Kuperberg, M., Huber, N., and Reussner, R. (2011). Ginpex: Deriving Performance-relevant Infrastructure Properties Through Goal-oriented Experiments. In *Proceedings of the 7th ACM SIGSOFT International Conference on the Quality of Software Architectures (QoSA 2011)*, pages 53–62, New York, NY, USA. ACM.

[Hauck et al., 2013] Hauck, M., Kuperberg, M., Huber, N., and Reussner, R. (2013). Deriving performance-relevant infrastructure properties through model-based experiments with ginpex. *Software & Systems Modeling*, pages 1–21.

[Herbst et al., 2013] 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 et al., 2014] 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.*

[Kounev et al., 2011a] Kounev, S., Bender, K., Brosig, F., Huber, N., and Okamoto, R. (2011a). Automated Simulation-Based Capacity Planning for Enterprise Data Fabrics. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 27–36, Brussels, Belgium, Belgium. ICST. Acceptance Rate (Full Paper): 29.8% (23/77), ICST Best Paper Award.

[Kounev et al., 2011b] Kounev, S., Brosig, F., and Huber, N. (2011b). Self-Aware QoS Management in Virtualized Infrastructures (Poster Paper). In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011).*

[Kounev et al., 2014] Kounev, S., Brosig, F., and Huber, N. (2014). Descartes Meta-Model (DMM). Technical report, Karlsruhe Institute of Technology (KIT). To be published.

[Kounev et al., 2010] 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 et al., 2012] Kounev, S., Huber, N., Spinner, S., and Brosig, F. (2012). Model-based techniques for performance engineering of business information systems. In Shishkov, B., editor, *Business Modeling and Software Design*, volume 0109 of *Lecture Notes in Business Information Processing (LNBIP)*, pages 19–37. Springer-Verlag, Berlin, Heidelberg.

[Thomas et al., 2011] Thomas, N., Bradley, J., Knottenbelt, W., Kounev, S., Huber, N., and Brosig, F. (2011). Preface. *Elec. Notes in Theoretical Computer Science*, 275:1 – 3.

Abstract

Recent trends like cloud computing and virtualization indicate that service providers are increasingly transitioning to IT infrastructures that provide higher flexibility to allocate resources dynamically in response to changes in the business landscape. These service providers are driven by the pressure to improve the efficiency of their systems, e.g., by sharing resources, to reduce their operating costs. To raise the resource efficiency, resource allocations must be continuously adapted during operation to changes in the system environment. However, the increasing system complexity and the rising frequency at which adaptations are required render human intervention prohibitive and increase the need for autonomic and self-adaptive approaches. Thus, many researchers in the autonomic computing and software engineering communities are working on approaches for the engineering of self-adaptive systems.

A powerful approach to tackle the challenges and manage the complexity of engineering such systems is to apply model-based techniques. The vital benefit of employing models for system adaptation is that models can provide relevant information for what-if analyses and thus drive the autonomic decision-making process. Thereby, it is possible to search for valid and suitable system configurations at the model level and thus avoid unnecessary and possibly costly system adaptations.

This thesis presents a systematic approach for proactive and autonomic performance-aware resource management in modern dynamic IT service infrastructures. Core of the approach is the Descartes Modeling Language (DML), a novel architecture-level modeling language specifically designed to describe performance and resource management related aspects in such environments. With DML it is possible to continuously predict at run-time the impact of changes in the system environment as well as the impact of potential adaptation actions on the system performance and resource efficiency. Furthermore, DML enables the specification of adaptation processes at the model level. These modeled processes are used to automatically adapt the system to changes in its environment, such as the way it is used by customers, leveraging DML's performance prediction capabilities for automated decision-making. By means of this model-based approach, it is possible to autonomically find suitable system configurations at the model level, thereby avoiding unnecessary and possibly costly system adaptations. In addition, the presented approach provides a method for self-adaptive workload forecasting to anticipate changes in the load placed on the system such that the system can be adapted *proactively*, i.e., before performance or resource efficiency problems occur. Thus, with our approach, substantial reductions in IT costs through significant improvements in the resource efficiency can be achieved. Moreover, the sophisticated modeling abstractions of DML support the model-driven development of self-adaptive systems and thereby help to reduce the inherent complexity of such tasks.

The scientific contributions of this thesis are:

- A novel modeling formalism to describe the complex nature of the resources of modern dynamic IT service infrastructures. The formalism, which is part of DML, supports modeling the distribution of resources within and across data centers as well

as the specification of the performance-influencing properties of these resources, including nested logical resource layers such as virtualization or middleware.

- A method for the identification, classification, and automated quantification of possible performance-influencing factors of resource layers, e.g., of virtualization platforms. The results can be used to derive a model of the performance overhead induced by a resource layer and thereby improve performance prediction results.
- A flexible modeling formalism to describe the degrees of freedom of architecture-level performance models that can be employed for run-time system adaptation. Furthermore, we provide modeling abstractions to specify the system adaptation processes at the architecture-level in an intuitive and easily maintainable manner. These abstractions can be used to describe high-level adaptation objectives, explicitly distinguishing system-specific adaptation actions from system-independent adaptation strategies.
- A novel method for self-adaptive workload classification and forecasting to enable proactive system adaptation at run-time. The method follows an approach that automatically identifies the characteristics of a given workload and selects a suitable forecasting strategy. The method continuously adapts the selection at run-time to reflect the characteristics of the observed workload, taking into account the achieved forecast accuracy as well as changing system requirements and user objectives.
- An end-to-end approach for autonomic performance-aware resource management leveraging the previous contributions to implement a holistic model-based adaptation control loop. The approach uses DML as the basis to support automated decision-making by predicting the impact of adaptation actions on the system performance and adapting the system accordingly. The proposed adaptation method also leverages the previously described models and techniques for proactive system adaptation.

From a technical perspective, we contribute a framework that implements the described concepts to automatically adapt the system model according to the corresponding modeled adaptation process, using performance prediction to evaluate the impact of its adaptation actions. We apply and evaluate our approach end-to-end in the context of three different representative case studies to demonstrate that our approach can be effectively used for autonomic performance-aware resource management to increase the resource efficiency in modern dynamic IT service infrastructures without sacrificing performance guarantees. The case studies are carefully selected and cover a broad spectrum of configurations with different types of applications, hardware environments, deployment configurations, and workloads. The considered evaluation scenarios are derived from typical real-life problems, e.g., of our industrial partner Blue Yonder, a leading service provider in the field of predictive analytics and big data. The results show that our approach can provide significant efficiency gains of more than 50% without sacrificing performance guarantees, and that it is able to trade-off performance requirements of different customers in heterogeneous hardware environments. Furthermore, the results show that our approach enables proactive system adaptation, reducing the amount of Service-Level Agreement (SLA) violations by 60% compared to trigger/rule-based approaches. These case studies and their results remarkably show how our holistic model-based approach can be effectively used to engineer systems with autonomic performance and resource management capabilities providing substantial reductions in IT costs.

Zusammenfassung

Trends wie Cloud Computing und Virtualisierung machen deutlich, dass immer mehr Dienstanbieter IT-Infrastrukturen einsetzen, die eine höhere Flexibilität und Dynamik bei der Zuweisung von Ressourcen bieten. So kann z.B. durch Teilen von Ressourcen die Effizienz der Systeme erhöht und damit die Betriebskosten gesenkt werden. Um die Ressourceneffizienz zu steigern, müssen aber die Ressourcenzuweisungen ständig während des Betriebs an Veränderungen in der Umgebung, wie z.B. Lastschwankungen, angepasst werden. Die manuelle Durchführung solcher Anpassungen werden jedoch erschwert durch die steigende Komplexität der Systeme sowie die zunehmende Frequenz, mit der Anpassungen nötig sind. Dies führt zu einer verstärkten Nachfrage nach autonomen und selbst-adaptiven Systemen, weshalb viele Forscher aus den Bereichen Autonomic Computing und Software Engineering an Methoden zur Konstruktion solcher Systeme arbeiten.

Eine Idee, die Herausforderungen und Komplexität, die bei der Entwicklung solcher Systeme auftreten zu bewältigen ist, modellbasierte Techniken zu verwenden. Entscheidender Vorteil bei der Verwendung von Modellen zur Adaption von Systemen ist, dass Modelle relevante Informationen für “Was wäre wenn”-Analysen liefern und dadurch einen autonomen Entscheidungsprozess vorantreiben können. Mit solch einem Ansatz können gültige und passende Systemkonfigurationen auf Modellebene gesucht und bewertet, und dadurch unnötige, möglicherweise teure Adaptionen des Systems vermieden werden.

Die vorliegende Arbeit stellt einen systematischen Ansatz zur proaktiven, autonomen und leistungsorientierten Ressourcenverwaltung in dynamischen IT-Infrastrukturen vor. Kern des Ansatzes ist die Descartes Modeling Language (DML), eine neue architekturbasierte Modellierungssprache, die speziell dafür entwickelt wurde, die für die Performance und die Ressourcenverwaltung relevanten Aspekte der Infrastruktur zu beschreiben. Mit DML ist es möglich, kontinuierlich zur Laufzeit den Einfluss von Änderungen in der Umgebung des Systems sowie den Einfluss von möglichen Adaptionen auf die Performance und Ressourceneffizienz des Systems vorherzusagen. Des Weiteren ermöglicht DML die Spezifikation von Adaptionsprozessen auf Modellebene. Diese modellierten Prozesse werden anschließend verwendet, um das System automatisch an Änderungen, wie z.B. im Benutzungsverhalten der Kunden, anzupassen. Dabei werden die Vorhersagefähigkeiten von DML zur autonomen Entscheidungsfindung herangezogen. Mit Hilfe dieses modellbasierten Ansatzes ist es möglich, auf Modellebene autonom passende Systemkonfigurationen zu finden und dadurch unnötige oder teure Systemadaptionen zu vermeiden. Zusätzlich bietet unser Ansatz die Möglichkeit zur Vorhersage der Arbeitslast, sodass Änderungen in der Last des Systems antizipiert und das System proaktiv, d.h. bevor Probleme mit der Performance oder Ressourceneffizienz auftreten, angepasst werden kann. Der vorgestellte Ansatz ist somit in der Lage, die Energieeffizienz des Systems signifikant zu verbessern und dadurch substantielle Einsparungen bei den IT-Kosten zu erzielen. Darüber hinaus unterstützen die zugeschnittenen Modellierungsabstraktionen von DML die modellgetriebene Entwicklung selbst-adaptiver Systeme indem sie helfen, die inhärente Komplexität dieser Aufgabe zu bewältigen.

Die wissenschaftlichen Kernbeiträge dieser Arbeit sind:

- Ein neuer Formalismus zur Modellierung des komplexen Charakters verteilter Ressourcen moderner IT-Infrastruktur. Als Teil der DML unterstützt der Formalismus bei der Modellierung der Verteilung von Ressourcen innerhalb und über Rechenzentren hinweg, sowie bei der Spezifikation der Performance-Einflussfaktoren der Ressourcen inklusive verschachtelter logischer Ressourcen wie z.B. von Virtualisierung oder Middleware.

- Eine Methode zur Identifikation, Klassifizierung und automatisierter Quantifizierung möglicher Performance-Einflussfaktoren von Ressourcenschichten wie z.B. Virtualisierungsplattformen. Die Ergebnisse können benutzt werden, ein Modell für den Performance-Overhead der Ressourcenschicht abzuleiten um damit die Performance-Vorhersage zu verbessern.

- Ein flexibler Formalismus zur Modellierung der Freiheitsgrade welche zur Laufzeit zur Anpassung des Systems zur Verfügung stehen. Des Weiteren bieten wir Modellierungsabstraktionen an, mit der auf Basis der Freiheitsgrade Prozesse zur Adaption des Systems bereits auf Architekturebene und in intuitiver und leicht zu wartenden Weise spezifiziert werden können. Diese Abstraktionen beschreiben die Ziele des Adaptionsprozesses auf hohem Abstraktionsniveau, und unterscheiden dabei explizit zwischen systemspezifischen Adaptionsaktionen und systemunabhängigen Adaptionsstrategien.

- Eine neue Methode zur selbst-adaptiven Klassifizierung und Vorhersage von Arbeitslast die zur proaktiven Systemadaption zur Laufzeit verwendet werden kann. Mit dieser Methode wird ein Ansatz verfolgt, der automatisch die Charakteristiken gegebener Lastkurven identifiziert und passende Vorhersagestrategien auswählt. Die vorgeschlagene Methode passt ihre Auswahl kontinuierlich zur Laufzeit an die Charakteristiken des beobachteten Lastverhaltens an und berücksichtigt dabei auch die erzielte Vorhersagegenauigkeit sowie veränderte Anforderungen an das System oder veränderte Zielvorgaben des Benutzers.

- Ein Ansatz zur autonomen leistungsorientierten Ressourcenverwaltung welcher auf die zuvor dargestellten Beiträge aufbaut und somit einen ganzheitlichen modellbasierten Regelkreis zur Systemadaption realisiert. DML dient dabei als Basis für die automatisierte Entscheidungsfindung indem die Auswirkungen von Adaptionsaktionen auf die Performance des Systems vorhergesagt werden und der Adaptionsprozess das System dementsprechend anpasst. Des Weiteren nutzt diese Adaptionsmethode die zuvor beschriebenen Modelle und Vorhersagetechniken zur proaktiven Systemadaption.

Aus technischer Sicht ist der Beitrag dieser Arbeit das Rahmenwerk, welches die beschriebenen Konzepte implementiert um automatisch das Modell des Systems anhand des modellierten Adaptionsprozesses anzupassen und dabei Performancevorhersagen verwendet, um die Auswirkungen von Adaptionen zu bewerten. Der vorgestellte Ansatz wird im Rahmen von drei repräsentativen Fallstudien im Ganzen evaluiert um zu demonstrieren, dass der Ansatz für autonome leistungsorientierte Ressourcenverwaltung verwendet werden kann um die Ressourceneffizienz in modernen dynamischen IT-Infrastrukturen zu erhöhen, ohne Performance-Garantien opfern zu müssen. Die Fallstudien wurden sorgfältig ausgewählt um ein möglichst breites Spektrum an Szenarien mit unterschiedlichen Arten von Hardwareumgebungen, Anwendungen, Softwareverteilungen und Arbeitslast abzudecken. Die betrachteten Evaluationsszenarien sind abgeleitet von typischen Problemen die in der Industrie auftreten, z.B. bei unserem Partner Blue Yonder, einem führenden Dienstanbieter

im Bereich Predictive Analytics und Big Data. Die Ergebnisse zeigen, dass unser Ansatz signifikante Effizienzsteigerungen von mehr als 50% ermöglicht, ohne Performance-Garantien zu opfern und dass er in der Lage ist, verschiedene Performance-Anforderungen unterschiedlicher Kunden abzuwägen, selbst in heterogenen Hardwareumgebungen. Des Weiteren zeigen die Ergebnisse, dass unser Ansatz die proaktive Adaption von Systemen unterstützt und dadurch im Vergleich zu trigger- oder regelbasierten Ansätzen die Anzahl der Verletzung von Dienstgütevereinbarungen um bis zu 60% senkt. Diese Fallstudien und die erzielten Ergebnisse zeigen deutlich, dass der vorgestellte ganzheitliche und modellbasierte Ansatz die systematische Konstruktion von Systemen mit Fähigkeiten zur autonomen Performance- und Ressourcenverwaltung ermöglicht und dadurch substantielle Senkungen der IT Kosten erzielt werden können.

1. Introduction

1.1. Motivation

Modern software systems have increasingly distributed architectures composed of loosely-coupled services that are typically deployed on virtualized infrastructures. Supported by virtualization, which abstracts from the physical infrastructure, such system architectures provide increased flexibility by enabling the dynamic assignment of virtual to physical resources at run-time. Recent trends like cloud computing confirm that more and more service providers adopt such dynamic infrastructures, driven by the pressure to improve the efficiency of their systems, e.g., by sharing resources, and to reduce their operating costs. For example, cloud computing allows to provide data center resources as on demand services over a private or public network in a pay-as-you-go manner. This promises substantial reductions in IT costs, as well as significant improvements in the energy efficiency through the dynamic consolidation of system resources and their sharing among multiple independent applications (Kaplan et al., 2008; IT world, The IDG Network, 2008).

However, to maintain performance requirements with improved resource efficiency, systems must be continuously adapted to changes in their environment such as workload fluctuations or added/removed services. For example, the amount of resources allocated to each service must be continuously adjusted to match the changing resource demands resulting from variations in the customer workloads. Ideally, such adaptations should be performed in an automated manner to reduce the amount of human intervention. The challenge that automated and autonomous resource management approaches are faced with is that the benefits of the increased flexibility and dynamics come at the cost of higher system complexity. The increased complexity results from the introduced gap between physical and virtual resource allocations and the complex interactions between the applications sharing the physical infrastructure. For example, changes in the workload behavior of one application can affect other applications if they are sharing resources or services. This complexity makes it difficult to apply trigger- or rule-based approaches as used, e.g., in the Amazon Elastic Compute Cloud (Amazon Web Services, 2010), because the appropriate triggering points are typically highly dependent on the architecture of the hosted services and on their workload profiles which can change frequently during operation. Moreover, trigger/rule-based approaches cannot know *in advance* how much additional resources in the various application tiers will be required (e.g., Virtual Machines (VMs), virtual CPUs of VMs, physical machines, network bandwidth) and where and how the newly started VMs should be deployed and configured to ensure performance requirements without sacrificing efficiency.

The difficulty to predict the effects of the described interactions between applications as well as the inherent semantic gap between application-level metrics and resource allocations at the physical and virtual layers significantly increase the complexity of managing the performance at system run-time. The pivotal question is: How can we perform system adaptations in such situations in an autonomic manner without disturbing the system operation? Thus, many researchers in industry and academia are working on techniques for designing and engineering *self-adaptive systems*; a system that is able to automatically adapt itself at run-time to changes in its environment to ensure its functionality, or Quality of Service (QoS) properties like performance and resource efficiency.

A promising approach to address the above described challenge in managing the complexity of modern dynamic IT service infrastructures is to develop self-adaptive systems with model-based adaptation mechanisms (Kramer and Magee, 2007; Blair et al., 2009). The essential advantage of a model-based approach to system adaptation is that, assuming that the employed system model reflects the system state with sufficient accuracy, the model can be used to ensure that the system will continue to operate as expected after a planned adaptation has been executed, taking into account the performance and resource efficiency requirements. Thus, by using models to search for suitable system configurations, unnecessary and sometimes costly adaptations of the system can be avoided. Furthermore, by abstracting from technical details, models help to reduce the inherent complexity of such systems and thereby support the model-driven development of self-adaptive systems. However, to achieve this goal, novel modeling formalisms are required that capture the structure, behavior, and goals of the system to support automated adaptation decisions at run-time.

The goal of the approach presented in this thesis is to provide a holistic model-based approach for autonomic performance-aware resource management leveraging novel modeling formalisms to support the automatic adaptation of resource allocations to changes in the system environment while maintaining performance requirements. Core of the approach is a novel architecture-level modeling language, called Descartes Modeling Language (DML), specifically designed to describe performance and resource management related aspects of modern dynamic IT service infrastructures. With DML it is possible to continuously predict at run-time the impact of changes in the way the system is used by customers, as well as the impact of changes in the system configuration, on the system performance and resource efficiency. Furthermore, DML can be used to specify adaptation processes at the model level in a human-understandable and reusable way. These processes are used to adapt the system during operation to the changes in its environment using DML's performance prediction capabilities for automated decision-making. In addition, the presented approach leverages workload forecasting techniques to adapt the system proactively, i.e., before the changes in the system environment actually have a negative impact on system performance or resource efficiency.

1.2. Problem Statement

Many researchers agree that a promising approach for engineering self-adaptive systems is the development of model-based adaptation mechanisms (e.g., Kramer and Magee, 2007; Blair et al., 2009; de Lemos et al., 2011; Oreizy et al., 1999; Garlan et al., 2004; Becker et al., 2012; Salehie and Tahvildari, 2009). To this end, novel modeling formalisms are needed that provide means to describe the various system properties that are relevant for automated adaptation decisions (such as structural, behavioral, or non-functional properties), while explicitly taking into account the dynamic context of the system (the degrees of freedom for adaptation, the various adaptation strategies, and the underlying goals driving the system adaptation). Furthermore, to support the systematic development of self-adaptive systems, flexible modeling abstractions to specify different types of adaptation processes, ranging from simple rules to complex heuristics or optimization algorithms, are needed.

The challenge when developing such modeling formalisms is that the latter must be specifically designed to provide suitable modeling abstractions for performance and resource management at system run-time (Blair et al., 2009). Furthermore, one must separate the possibly multiple concerns that should be covered by the modeling formalism (France and Rumpe, 2007). Specifically, for the approach presented in this thesis, we need a modeling formalism that abstracts from technical and system-specific details to reduce complexity such that adaptation processes can be specified at the architecture-level in a human-understandable, machine processable, and reusable manner. At the same time, the modeling formalism should provide means to model sufficient performance-relevant details of the system to predict the performance impact of changes in the system environment and of undertaken adaptation actions. Thus, to provide novel modeling concepts for autonomic performance-aware resource management, we face the following main challenges and research questions.

Modeling abstractions for describing dynamic IT service infrastructures

The semantic gap between physical and virtual resource allocations in modern dynamic IT service infrastructures makes it hard to predict the effect of dynamic changes in the environment, as well as to predict the effect of possible reconfiguration actions on the system performance. To enable such predictions, a novel modeling formalism is required to capture the performance-relevant factors of modern dynamic IT infrastructures, focused on information that can be leveraged during system adaptation for automated decision-making. For example, the formalism must be capable of capturing the performance influences of the physical and the logical layers of the system architecture, so that their impact on the system performance for a given system configuration and workload scenario can be predicted. Specific research questions in this area are: What is a suitable abstraction level for modeling the performance-relevant properties and resource management aspects of the system that are relevant for automated adaptation decisions? How can we encode important structural information about the distribution of physical and logical resources? What are suitable modeling abstractions to describe the performance influences of multiple resource layers and how can these influences be quantified in an automated manner?

Modeling abstractions to specify system adaptation

To ease the development of self-adaptive systems, novel modeling formalisms are needed that support the specification of system adaptation processes at the model level. A modeling formalism must provide means to capture the dynamic aspects of the system, such as the degrees of freedom in which the system can be adapted at run-time. Moreover, the modeling formalism must support a human-understandable way of specifying how the system adapts to changes in its environment such that its operational goals are continuously fulfilled. Simultaneously, the modeled adaptation processes must be machine-processable to leverage model-driven techniques for executing the modeled adaptation process. Research questions that arise are: How can we reflect the adaptable parts of the system at the model level? How can we abstract from system-specific implementation details to describe adaptation processes in a generic, human-understandable and reusable way?

Besides these modeling formalisms, to proactively adapt a system to changes in the application workloads, we need techniques to forecast future workload variations such as growing workload intensity, load fluctuations, or periodic spikes. We can then use performance prediction techniques to predict the impact of the forecast workload variations on the system performance and perform proactive system adaptation in case of detected problems. The challenges and research questions in this context are:

Self-adaptive workload forecasting at system run-time

The proactive adaptation of systems to changes in their workloads requires novel workload forecasting methods designed for use during system operation that can automatically capture the observed workload trends and seasonal behavior. To provide accurate and accurately timed forecasts with an acceptable computational overhead for possibly multiple different workloads, the method must be capable of selecting at run-time the appropriate forecasting strategy depending on the observed characteristics of the workloads. When selecting a suitable forecasting strategy, the selection algorithm must trade-off between forecast accuracy and system-specific time constraints in which the forecasts must be available. Furthermore, the method must be able to continuously adapt its decision depending on changes of the workload characteristics, system requirements, or user objectives. Specific research questions to be addressed are: How can we automatically select a suitable workload forecasting strategy for a given workload? How can we consider system-specific requirements and user-defined objectives in the selection process? How to trade-off forecasting accuracy and forecasting speed?

Finally, the overall challenge is to demonstrate that the developed concepts can be effectively used to implement model-based system adaptation processes capable of autonomically managing the system's operational goals. Therefore, the developed modeling and prediction techniques must be consolidated into a coherent, clearly defined process model that leverages the described modeling formalisms for online performance prediction and autonomic decision-making. The basis for such a process model is typically a feedback loop (or control loop) which is considered as essential building block for engineering self-adaptive systems (Brun et al., 2009). The question remains what type of feedback loop should be used and how the individual parts can be integrated into the different phases of the loop.

1.3. Shortcomings of Existing Approaches

State-of-the-Art in Industry

The state-of-the-art of industrial approaches for automated performance and resource management in virtualized environments generally follow a trigger/rule-based approach when it comes to enforcing Service-Level Agreements (SLAs). Custom triggers can be configured that fire when a metric reaches a certain threshold (e.g., high resource utilization or load imbalance) and execute certain predefined reconfiguration actions until a given stopping criterion is fulfilled. The most prominent examples of such approaches are the Amazon Elastic Compute Cloud that offers "auto scaling" (Amazon Web Services, 2010), the "Autoscaling" solution of the Windows Azure technology platform (Microsoft, 2012), or the VMware Distributed Resource Scheduler that dynamically allocates and balances computing capacity (VMware, 2006).

The problem of these approaches is that application-level metrics (such as response time) normally exhibit a non-linear behavior on system load and they typically depend on the behavior of multiple VMs across several application tiers. Generally, it is hard to predict how changes in the application workloads (e.g., varying request arrival rates and/or transaction mix) propagate through the layers and tiers of the system architecture down to the physical resource layer. Therefore, it is hard to determine general thresholds indicating when triggers should be fired given that such triggers are typically highly dependent on the architecture of the hosted services and on their workloads which can change frequently during operation. Furthermore, in case of contention at the physical resource layer, the performance of an individual application may be significantly influenced by applications running in other co-located VMs sharing the physical infrastructure. Moreover, triggers are usually defined by the customer, i.e., the resulting adaptations are limited to the concerns a single customer. Thus, it is not possible to trade-off the requirements of multiple customers when trying to improve resource efficiency. To be effective, triggers must take into account the interactions between applications and workloads at the physical resource layer. The complexity of such interactions and the inability to predict how changes in application workloads propagate through the layers of the system architecture down to the physical resource layer render conventional trigger-based approaches unable to reliably enforce SLAs in an efficient and proactive fashion.

State-of-the-Art in Academia

In academia, there are two major lines of research trying to address the described problem from different perspectives. First, there is the (software) performance engineering community focusing on approaches for using performance models for capacity planning at run-time. Existing work in this area mainly uses coarse-grained performance models that typically abstract systems and applications at a high level (e.g., Jung et al., 2010; Zhang et al., 2007; Li et al., 2009; Bennani and Menasce, 2005; Cunha et al., 2007; Gambi et al., 2013). Such models do not explicitly model the software architecture and execution environment to distinguish performance-relevant behavior at the virtualization level vs. at the level of applications hosted inside the running virtual machines. The individual effects and complex interactions between the application workloads and the system components and layers are considered as static and viewed as a black box in such models. This hinders fine-grained performance predictions that are necessary for efficient resource management, e.g., predicting the effect on the response times of different services, if a virtual machine in a given application tier is to be replicated or migrated to another host, possibly with a different configuration; or predicting the effect of changing a configuration parameter in the virtualization or middleware layer of a given application tier (e.g., number of virtual CPUs assigned to an application server VM).

Over the last decade, the software performance engineering community has also proposed a number of modeling approaches for building architecture-level performance models of software systems (Koziolek, 2010). Such models provide modeling constructs targeted at describing and evaluating the performance-relevant behavior of a software system during the software development process, i.e., at system design-time and in an offline setting. However, these modeling constructs are unsuitable for system adaptation in online settings as there are some fundamental differences between offline and online scenarios that lead to different requirements on the underlying performance abstractions. 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 performance modeling abstractions should enable predicting the impact of changes in the application workloads as well as adaptation actions undertaken during operation to avoid performance issues. However, currently, there are no performance modeling approaches that explicitly consider the dynamic aspects of modern IT systems and services as part of their architectural models (Becker et al., 2012).

The second research area relevant to the approach presented in this thesis is represented by the autonomic computing and self-adaptive systems community. In this research area, models play an important role in managing the complexity of dynamic and self-adaptive systems and supporting adaptation decisions in such environments (Blair et al., 2009;

Cheng et al., 2009). As surveyed by Salehie and Tahvildari (2009), many approaches in this area use models to capture context information that can be leveraged during system adaptation. However, such approaches typically apply domain-specific models that abstract from the system architecture. Approaches that employ software architecture models are, e.g., Oreizy et al. (1999); Floch et al. (2006); Garlan et al. (2009); Hallsteinsen et al. (2012). However, such approaches are typically restricted to parameter and component composition adaptations and do not include the system's operational environment within the scope of the considered adaptation possibilities. Furthermore, adaptation decisions are normally based on simple policies or rules that do not consider detailed predictions of the impact of possible adaptation actions on the end-to-end system performance. Thus, the challenge that has not yet been fully covered by research in the autonomic computing and self-adaptive systems community is how to model the system performance properties and behavior as well as the adaptation processes at the system architecture-level in a human-understandable, reusable and machine processable manner.

1.4. Contributions of this Thesis

The contribution of this thesis is the design and implementation of a holistic model-based approach for autonomic performance-aware resource management of modern IT service infrastructures. In this approach, we use specifically designed architecture-level performance models for predicting the performance impact of changes in the environment (including both external changes in application workloads and internal changes as part of adaptation actions) as well as models designed to specify adaptation processes of such systems at the architecture-level.

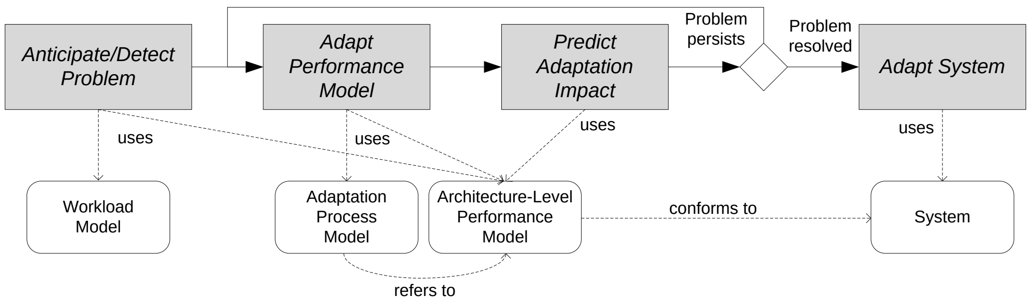


Figure 1.1.: Schematic representation of the model-based adaptation process and the involved artifacts.

From a high-level perspective, our model-based system adaptation process consists of the steps depicted in Figure 1.1. First, in the *Anticipate/Detect Problem* step, we apply workload forecasting techniques to predict changes in the application workloads. These forecasts serve as input to the architecture-level performance model of the system, which allows us to predict the impact of such changes on the system performance. If the workload changes have a negative impact on the system performance or resource efficiency, we use the adaptation process specified by the adaptation process model to adapt the system model (*Adapt Model*). In the *Predict Adaptation Impact* step, we predict the impact of the performed adaptation. If the applied adaptation was successful, i.e., it solves the problem that necessitated the adaptation, we adapt the system accordingly (*Adapt System*). If the problem is not solved, we repeat these steps until we find a model configuration that solves the detected problem.

The benefit of this approach is that we can search for solutions to detected or anticipated problems at the model-level and thereby avoid possibly costly adaptation actions on the real system. The performance model can be leveraged during the adaptation process to evaluate the impact of the possible adaptation actions and thereby support automated decision-making. To derive performance metrics from the architecture-level performance model, we use the online performance prediction techniques developed by Brosig (2014). More specifically, online performance prediction is used in the *Anticipate/Detect Problem* step to predict the impact of the workload changes on the system performance as well as in the *Predict Adaptation Impact* to evaluate the impact of adaptation actions. Hence, this thesis is closely related to the PhD thesis of Brosig as it leverages the online performance prediction techniques presented therein.

The contributions of this thesis can be separated into scientific and technical contributions. The scientific contributions are:

- A novel modeling formalism to describe the increasingly complex nature of modern distributed IT service infrastructures. The formalism supports modeling the distribution of resources within and across the boundaries of data centers as well as the specification of performance-relevant properties of the modeled resources. Furthermore, the modeling formalism also provides means to specify the performance influences of nested logical resource layers, such as virtualization or middleware. The proposed modeling formalism has been published in Huber et al. (2012a).

- A method for the identification, classification, and automated quantification of possible performance-influencing factors of resource layers, using virtualization as proof-of-concept. Based on the classification, we conduct automated benchmark experiments to quantify the influences of the identified factors. The automated quantification method has been published in Huber et al. (2010b). The derivation of performance overhead factors and the overall evaluation of the proposed approach has been published in Huber et al. (2011b).

- A flexible modeling formalism to describe the dynamic aspects of self-adaptive systems as well as their adaptation processes. This modeling formalism consists of two parts: modeling abstractions for specifying adaptation points and a modeling language to define adaptation processes. Adaptation points describe the degrees of freedom of the system architecture and valid configuration space. The adaptation process modeling language is designed to describe adaptation processes of self-adaptive systems at the architecture-level in an intuitive and easily maintainable manner. It distinguishes high-level adaptation objectives from low-level implementation details, explicitly separating system-specific adaptation operations from system-independent adaptation plans. The adaptation points meta-model has been published in Huber et al. (2012a). The adaptation process modeling language has been initially published in Huber et al. (2012b) and refined in Huber et al. (2014).

- A novel method for self-adaptive workload classification and forecasting providing the basis for *proactive* system adaptation at run-time. For a given workload, our method automatically identifies the characteristics of the workload and selects a suitable forecasting strategy. At run-time, the method continuously adapts the selection, taking into account the achieved forecast accuracy as well as changing workload characteristics or changing system requirements and user objectives. This method has been published in Herbst et al. (2013a) and refined in Herbst et al. (2014).

- A process model for using architecture-level performance models for self-adaptive resource allocation (Huber et al., 2011a), realizing a holistic model-based adaptation control loop for autonomic performance-aware resource management. Based on the familiar generic control loop used in the autonomic computing and software engineering communities, we define an adaptation method that leverages the previously described models and performance prediction techniques for proactive system adaptation. To support automated decision-making, we use models to predict the impact of adaptation actions on the system performance and adapt the system accordingly.

From a technical perspective, we contribute a framework that implements the described contributions to automatically adapt the system according to the corresponding modeled adaptation process, using online performance prediction to evaluate the impact of its adaptation actions. To demonstrate that the presented approach can be effectively used for autonomic performance-aware resource management, we apply and evaluate our approach end-to-end in the context of three different representative case studies. The considered evaluation scenarios are derived from typical real-life problems, e.g., of our industrial partner Blue Yonder, a leading service provider in the field of predictive analytics and big data. The case studies are carefully selected and cover a broad spectrum of configurations with different types of applications, hardware environments, deployment configurations, and workloads. The first case study is based on the SPECjEnterprise2010 benchmark that models a representative enterprise application and demonstrates the effectiveness of our approach in a homogeneous virtualized cluster environment. The second case study evaluates the potential of our workload classification and forecasting approach for proactive model-based system adaptation at run-time. The third case study is conducted in cooperation with Blue Yonder and investigates the applicability of our approach in heterogeneous environments built from commodity hardware to trade-off different performance requirements of multiple customers while maintaining resource efficiency.

The validation results show that our approach can provide significant resource efficiency gains of over 50% without sacrificing performance guarantees, and that it is able to trade-off performance requirements of different customers in heterogeneous hardware environments. Furthermore, we demonstrate how our approach enables proactive system adaptation and thereby considerably reduces the amount of SLA violations by 60% compared to trigger/rule-based approaches. With these end-to-end case studies, we demonstrate that i) DML allows the specification and execution of proactive system adaptation processes at the model-level to achieve significant performance and resource efficiency gains, and ii) architecture-level performance models and online performance prediction can be effectively used to perform autonomic system adaptation such that the system's performance requirements and resource efficiency targets are maintained. The versatility of the selected case studies illustrates that our approach can be applied not only in classical data centers but also in other service infrastructures such as private or public clouds. Thus, we consider the approach presented in this thesis as the foundation for a new type of computing systems which are designed from the ground up with built-in online QoS prediction and self-adaptation capabilities used to enforce QoS requirements in a cost- and energy-efficient manner.

1.5. Outline

The thesis is structured into three main parts. Part I describes the foundations of this thesis and discusses related work. It is organized as follows.

Chapter 2 presents the foundations of this thesis. The chapter introduces common concepts and terminology from the area of autonomic computing, self-adaptive systems, and performance engineering, and explains the role of models in these areas.

Chapter 3 reviews related work in the areas of autonomic computing, self-adaptive systems, and performance engineering, focusing on architecture-based approaches to self-adaptation.

Part II comprises the contributions of this thesis. The contributions are separated into the following four chapters.

Chapter 4 presents the major conceptual building blocks of our model-based adaptation approach. More specifically, it presents a refined concept of the MAPE-K adaptation control loop designed to leverage the novel features of the Descartes Modeling Language (DML) to realize autonomic performance and resource management at run-time. Moreover, Chapter 4 introduces the major concepts of DML which we need for online performance prediction and run-time system adaptation.

Chapter 5 presents a meta-model to describe the resource landscape of modern distributed IT systems. This chapter introduces novel concepts for modeling the configuration of physical resources as well as the performance-influencing properties of resource layers. Furthermore, it presents a method for the automatic quantification of such performance-influencing properties using virtualization as proof-of-concept.

Chapter 6 introduces generic and flexible modeling formalisms to describe the dynamic aspects of self-adaptive systems as well as their adaptation processes. It also presents the architecture of the adaptation framework that implements the presented concepts of our model-based adaptation control loop.

Chapter 7 presents an approach for self-adaptive Workload Classification and Forecasting (WCF) at run-time that uses common time series analysis techniques to identify the characteristics of workloads. Based on the identified characteristics, suitable forecasting methods to predict future workload intensities are selected and used for proactive system adaptation.

Each of these chapters contain their own evaluation in which we evaluate the individual contributions in isolation. Part III presents the end-to-end validation scenarios and results and concludes the work.

Chapter 8 presents the end-to-end validation of our approach showing how the individual parts of our approach can be integrated into a holistic model-based approach that can be applied for autonomic performance-aware resource management. We conduct three case studies with different validation goals and discuss the results as well as the threats to validity.

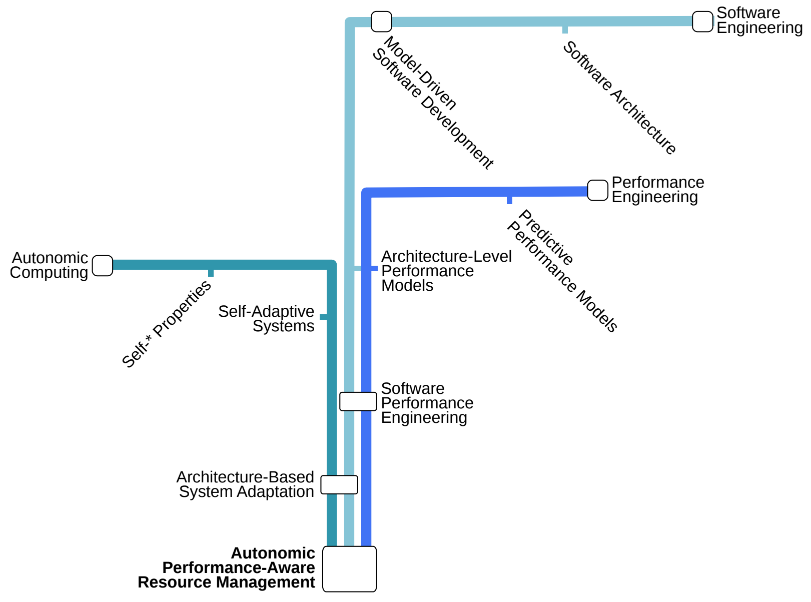
Chapter 9 concludes this thesis by summarizing its contributions and validation results and gives an outlook on future research.

Part I.

Foundations and Related Work

2. Autonomic System Adaptation and Performance Modeling

The approach presented in this thesis is influenced by three major lines of research, depicted in Figure 2.1: autonomic computing, software engineering, and performance engineering. This chapter introduces the foundations of this thesis and explains the most important concepts and terminology. In Section 2.1, we introduce foundations of autonomic computing, focusing on the major self-* properties and design principles of autonomic and self-adaptive systems. In Section 2.2, we present concepts from software engineering and performance engineering that are used in the field of software performance engineering. More specifically, we discuss different types of performance models and their properties that are used during software development as well as at system run-time for autonomic performance-aware resource management. In Chapter 3, we review related work from the area of model-based performance and resource management (Section 3.2) and architecture-based system adaptation (Section 3.1).



Software Engineering
Model-Driven Software Development
Software Architecture
Performance Engineering
Predictive Performance Models
Autonomic Computing
Architecture-Level Performance Models
Self-* Properties
Self-Adaptive Systems
Software Performance Engineering
Architecture-Based System Adaptation
Autonomic Performance-Aware Resource Management

Figure 2.1.: Related research areas.

2.1. Autonomic and Self-Adaptive Systems

The term *autonomic computing* was first introduced by IBM in 2001 (Horn, 2001). It proposes a biology-inspired notion of a computer system that is able to adapt to internal and external changes with minimal human intervention (Kephart and Chess, 2003). This paradigm was motivated by the problem that IT systems were becoming increasingly complex and thus difficult for human administrators to manage and maintain. As a result, many researchers and developers began to examine ways to overcome these problems by aiming to automate administrative tasks, such as updating or reconfiguring the system. This led to the idea of enhanced computing systems with self-management capabilities that are able to evolve in an autonomous manner, detecting and fixing undesirable behaviors and adapting to changes in their environment.

Autonomic computing is inspired by research from many other fields, such as biology, control theory, artificial intelligence, complex systems, etc. For example, the term autonomic is inspired by the autonomic nervous system responsible for regulating vital functions in the complex system of the human body, and feedback loops known from control systems are also a central element of autonomic computing systems. Moreover, autonomic systems are related to artificial intelligence as both types of systems seek to achieve certain operational goals with little or no human intervention. More details about the relation of autonomic computing to other research areas can be found in Lalanda et al.

In the following sections, we first present the so-called self-* properties that are targeted by autonomic or self-adaptive systems. Next, we present the major concepts and principles important for engineering autonomic and self-adaptive systems. Then, we discuss the role of models during the development and execution of autonomic and self-adaptive systems. Finally, we present a definition for self-aware computing systems, a sub-class of autonomic computing systems.

2.1.1. Self-* Properties

According to Kephart and Chess (2003), the self-management capabilities of an autonomic computing system can be separated into four main areas:

1. Self-configuration: The system adapts to unpredictable conditions by automatically changing its configuration, e.g., adding or removing new components or resources, without disrupting its service.
2. Self-healing: The system can prevent and recover from failure by automatically discovering, diagnosing, circumventing, and recovering from issues that might cause service disruptions.
3. Self-optimization: The system is able to continuously tune itself either reactively or proactively in response to changes of environmental conditions.
4. Self-protection: The system anticipates, identifies and prevents various types of threats in order to preserve its integrity and security.

These four properties are considered as fundamental for any autonomic system, and are also referred to as *self-chop* properties. Since the beginning of the autonomic computing initiative, further attributes and capabilities have been identified by researchers in this area. Such self-managing properties (sometimes referred to as self-* properties) are interrelated properties that a system should possess in order to achieve various degrees of autonomicity. For example, other important self-* properties according to Lalanda et al. (2013) are:

- Self-predicting (self-anticipating): a system's ability to predict future events or requirements, whether with respect to the system's internal behavior or with respect to its external context. An anticipating system should be able to manage itself proactively.
- Self-adapting: a system's ability to modify itself (self-adjust) in reaction to changes in its execution context or external environment, in order to continue to meet its business objectives despite such changes.
- Self-adjusting: a system's ability to modify itself during run-time including modifications to its internal structure, configuration, or behavior.
- Self-aware: a system's ability to 'know itself', i.e., to possess knowledge of its internal elements, their current status, history, capacity, and connections to external elements or systems. A system may also possess knowledge of the possible actions it may perform and of their probable consequences. This property can also be considered as a combination of self-reflecting, self-predicting, and self-adapting, and is an important property for the approach presented in this thesis. We will discuss it in more detail in Section 2.1.4.
- Self-configuring: a system's ability to (re-)configure itself by (re-)setting its internal parameter values to achieve high-level policies or business goals.
- Self-diagnosing: a system's ability to analyze itself in order to identify existing problems or to anticipate potential issues.
- Self-governing (self-managing): a system's ability to administer itself in order to achieve high-level policies or business goals.
- Self-healing (self-repairing): a system's ability to recover from the failure of any of its constituent elements (reactive) or to predict and prevent the occurrence of such failures (proactive).
- Self-monitoring: a system's ability to retrieve information on its internal state and behavior, whether globally or for any of its constituent elements. Self-monitoring is essential for attaining self-awareness.
- Self-optimizing: a system's ability to improve its operation with respect to predefined goals (e.g., resource management for optimized system efficiency).
- Self-organized (self-assembled): a system's property of being automatically formed via the decentralized assembly of multiple independent elements.
- Self-protecting: a system's ability to protect itself from malicious or inadvertent attacks.
- Self-reflecting: a system's ability to reflect its software architecture, execution environment, and hardware infrastructure on which it is running as well as its operational goals (e.g., performance and resource efficiency targets, or other QoS requirements). Sometimes this property is also referred to as the system's ability to determine whether its self-* functionalities conform to expectations.
- Self-stabilizing: a system's ability to attain a stable, legitimate state, starting from an arbitrary state and after a finite number of execution steps.

The described self-* properties typically influence each other to a certain degree and thus, they normally cannot be considered in isolation. They must be considered from a global system perspective and the integration and coordination of such properties within the whole computing system requires a systematic approach (Lalanda et al., 2013). In the following, we present the major design concepts and a reference architecture currently used in autonomic and self-adaptive systems to implement systems providing one or multiple of the described properties.

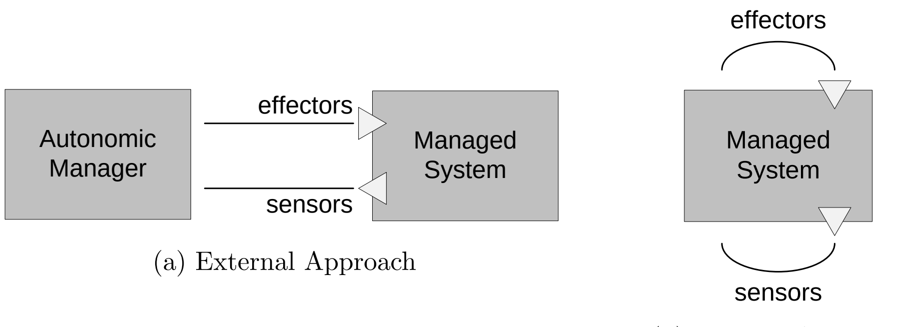
2.1.2. Design Principles of Autonomic Systems

Core of autonomic systems is an entity that realizes the self-* properties presented in the previous section (IBM Corporation, 2003). This entity, also referred to as *autonomic manager* in the following, can be understood as an executable software unit that implements the adaptation logic in order to continuously meet the system's operational goals.

Autonomic Manager and Managed System

In their taxonomy on self-adaptive software systems, Salehie and Tahvildari (2009) distinguish two types of approaches to implement autonomic managers (cf. Figure 2.2). First, the *external approach* distinguishes a dedicated adaptation engine (or manager) that implements the logic for adapting the managed system. The managed system can be a single server or a cluster of machines in a Grid environment, a specific software component, an operating system or a component therein, etc. Conceptually, the managed system provides two types of control points: sensors and effectors. Sensors provide information about the managed system, e.g., information about its state or its current performance. Effectors provide the possibility to adjust the managed system.

In certain cases, the autonomic manager and the managed system may be more intertwined without a clear separation of application logic from adaptation logic. Such *internal approaches* are normally based on programming language features, such as conditional expressions, parametrization, or exceptions.



(a) External Approach

(b) Internal Approach

Figure 2.2.: External and internal approaches to implement autonomic managers.

Autonomic Control Loop

The adaptation behavior realized by autonomic managers typically corresponds to the concept of a control loop. Such control loops can be found in other research areas too, e.g., in feedback control systems used in control theory (cf. Brogan, 1991). In a feedback control system, the output of the system is fed back through a sensor to a reference value (cf. Figure 2.3). The controller then takes the error (or difference) between the reference value and the output to change the input values to the system under its control.

To structure the principle of operation exhibited by autonomic managers, Kephart and Chess (2003) defined a reference architecture that is also based on a control loop, typically referred to as the MAPE-K loop. This reference architecture has the advantage that it offers a clear way to identify and classify areas of particular focus and thus, it is used by many researchers to communicate the architectural concepts of autonomic systems.

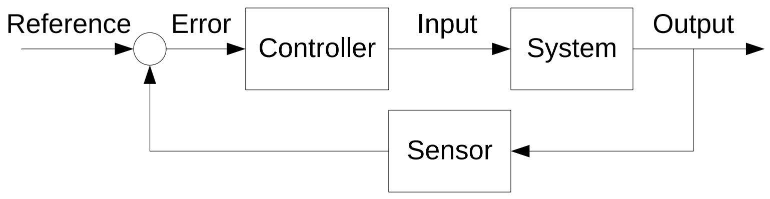


Figure 2.3.: Feedback loop in control systems.

The acronym MAPE-K reflects the five main constituent phases of autonomic loops, i.e., MONITOR, ANALYZE, PLAN, EXECUTE, and KNOWLEDGE, as depicted in Figure 2.4. Basically, the MONITOR phase collects information from the sensors provided by the managed artifacts and its context. The ANALYZE phase uses the data of the MONITOR phase to assess the situation and determine any anomalies or problems. The PLAN phase generates an adaptation plan to solve a detected problem. The EXECUTE phase finally applies the generated adaptation plan on the actual system. A cross-cutting aspect shared among all phases of the loop is the KNOWLEDGE about the system and its context, capturing aspects like the software architecture, execution environment, and hardware infrastructure on which the system is running. The knowledge may also explicitly capture the *operational goals* of the system, e.g., the target QoS level the managed system should provide. The representation of the knowledge can take any form, e.g., a performance model describing the performance behavior of the system. As this thesis follows a model-based approach, we are particularly interested in the different types of models that can be applied to represent the knowledge about the system. Such concepts will be introduced in the following Section 2.1.3.

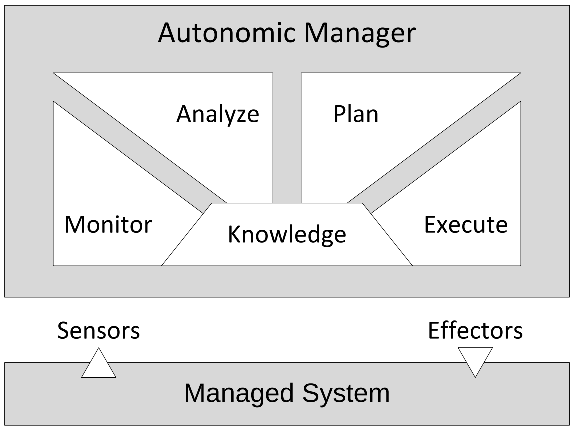


Figure 2.4.: Reference architecture of the autonomic control loop implemented by autonomic managers (cf. Kephart and Chess, 2003).

The software engineering community uses a similar feedback loop concept, distinguishing the four phases COLLECT, ANALYZE, DECIDE, and ACT (Cheng et al., 2009). Conceptually, the behavior of these phases is similar to the phases in the MAPE-K loop, however, this concept does not explicitly consider the KNOWLEDGE part.

More details about the use of feedback loops in self-adaptive systems, such as the use of multiple, multi-level, positive, or negative feedback loops, are given by Brun et al. (2009).

Reactive and Proactive Autonomic Managers

We distinguish two different general types of adaptation behavior that are implemented by an autonomic manager, the reactive and proactive adaptation behavior (Lalanda et al., 2013). The reactive behavior triggers the adaptation only in reaction to some external event, such as availability of new monitoring data or violations of operational goals. In contrast, a proactive manager may take the initiative of analyzing future situations to anticipate possible problems and adjust the managed system before the anticipated problems actually occur.

Structuring Multiple Autonomic Managers

Autonomic systems may consist of a number of autonomic elements that manage parts of the global autonomic system. Normally, the decoupling is domain specific and can depend on multiple architectural or functional considerations. Although the different autonomic elements can act independently and with no dependency between their respective management actions, they usually cooperate to achieve common goals. There exist various collaboration patterns for organizing multiple autonomic elements.

One possible approach to multi-agent cooperation is a *hierarchical* (centralized) structuring of the autonomic elements. Autonomic elements are organized into a hierarchy where elements can set goals to the elements of lower levels in the hierarchy, which in turn provide feedback about their behavior. An alternative to the hierarchical approach is where all autonomic elements in the system can communicate directly (the *decentralized* approach). In this organization pattern, each autonomic element acts independently and uses both external and internal data to make its own decisions. The particular challenge of this approach lies in guaranteeing that the behavior emerging from the individual goals of each autonomic element will actually contribute to reaching the global goal.

2.1.3. Model-Driven Autonomic and Self-Adaptive Systems

When it comes to the Knowledge part of the reference architecture presented in Section 2.1.2, the crucial question is: What should be stored in the knowledge and how should it be represented? A promising approach to represent the knowledge is to use models that abstract from the complexity of the managed system and that allow to reason about the system properties. For example, a great benefit of using models for adaptation planning is that, under the assumption that the model mirrors the managed system with sufficient accuracy, the model can be used to ensure that system integrity is preserved when applying an adaptation. This is because changes are planned and applied to the model first, allowing to identify the resulting system state including any violations of constraints or requirements captured as part of the model. The usage of explicit models as Knowledge allows the MAPE tasks to focus on the computation (the adaptation logic) and not on the data representation and collection. In general, many different types of models can be used. However, it is important to use models defined at the right level of abstraction. For example, models abstracting the system at the architecture level can provide benefits for the systematic engineering of autonomic and self-adaptive systems (cf. Kramer and Magee, 2007; France and Rumpe, 2007; Blair et al., 2009). In the following, we discuss different types of models that play an important role as artifacts that are used in software engineering along the software life cycle.

The model-driven engineering (MDE) community advocates the creation and exploitation of models to entirely drive the development and maintenance of software systems. To be machine-processable, models must be formalized using a modeling language with clear, precise and non-ambiguous semantics. Since all aspects of such a modeling language have to be expressed and formally defined, the modeling language itself can be seen as a model (i.e., a *meta-model*). This meta-model can be considered as a model that defines the language for building a concrete system model and is thus sometimes also called domain-specific language (DSL). Basically, a meta-model or a DSL defines a grammar and a vocabulary that allows to create models that conform to the meta-model.

In recent years, models are also seen as crucial artifacts that can play an important role during the whole software life cycle and not only at design time. France and Rumpe (2007) introduced the idea of models at run-time (*models@runtime*) to capture and describe run-time phenomena in an abstract manner. Such models are synchronized with (or causally connected to) an operational system and can be used to reflect information about the system operations. This notion of models@runtime has been refined by Blair et al. as a "causally connected self-representation of the managed system that emphasizes the structure, behavior, or goals of the system from a problem space perspective" (Blair et al., 2009, p. 2). As such, models@runtime are an important contribution to the field of autonomic computing as they provide the relevant information to drive autonomic decision-making.

When developing modeling formalisms, one is usually faced with the problem that models should cover multiple concerns (cf. France and Rumpe, 2007). Thus, the model that represents the managed system usually consists of several models focusing on different aspects, possibly at different levels of abstraction. Vogel et al. (2011) present a classification of the different types of models that can constitute a run-time model (cf. Figure 2.5). First, they distinguish two types of models: reflection models and adaptation models. *Reflection models* reflect the system and its environment either in a descriptive or prescriptive manner (system models and environment models, respectively). Descriptive models describe the "as-is" situation of the running system and its environment, while prescriptive models prescribe the "to-be" situation, i.e., the designated target state of the system. The system and/or environment models are either analyzable themselves, or they serve as input for analysis models to support reasoning about the system and/or its environment (Vogel et al., 2011).

In contrast to reflection models, *adaptation models* are primarily applied on reflection models and they define how reflection models are evaluated or changed. The reasoning performed on a reflection model is specified through *evaluation models*, e.g., by specifying constraints that are checked on the reflection model. A *change model* describes how to explore the system's variability and configuration space to find a suitable adaptation plan (Vogel and Giese, 2012). For example, a reflection model may describe the architecture and performance-relevant aspects of the managed system, whereas adaptation models would describe how to evaluate the performance model (evaluation model) and change it in case adaptations are required (change model).

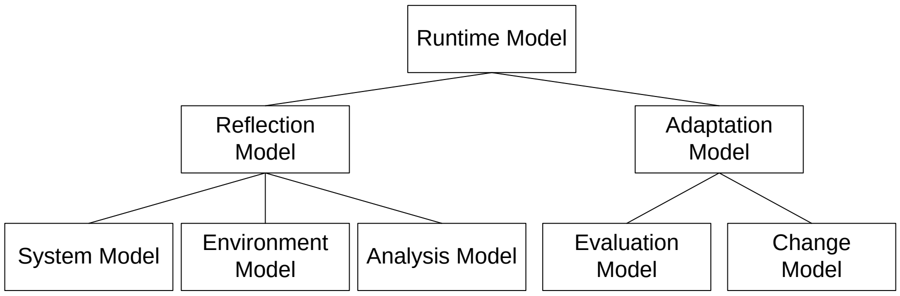


Figure 2.5.: Classification of different model types that constitute a run-time model according to Vogel et al. (2011).

Building a run-time model of the managed system is a non-trivial task. One specific challenge is to find a suitable abstraction level that allows to model the system components, their interactions, and their behavior with sufficient accuracy. Furthermore, the abstraction level should also support automated decision-making and enable the specification of adaptation processes at the model level.

Modeling adaptation at the architectural level is a popular approach (e.g., Kramer and Magee, 2007; Oreizy et al., 1999; Garlan et al., 2004; Floch et al., 2006). *Architecture-level* (run-time) models describe the system's structure and behavior at the level of the software architecture.Models may also capture some aspects of the operating environment in which the managed elements are deployed and they may also describe the operational goals of the managed system. More details about different types of architecture- and model-based approaches for autonomic and self-adaptive systems will be presented and discussed in Chapter 3 as part of related work.

2.1.4. The Descartes Research Project and Self-Aware Computing Systems

In the 17th century, the French philosopher and mathematician René Descartes described the bidirectional link between the mind and the body as the *dualism principle* ("the mind controls the body, but the body can also influence the mind"), sometimes paraphrased in his famous words "cogito, ergo sum" (Descartes, 1644).

The Descartes Research Projectthat funded the approach presented in this thesis, pursues the vision of *self-aware computing systems* that have built-in online QoS prediction and self-adaptation capabilities to address the challenges of autonomic performance and resource management (Kounev, 2011). In this context, self-aware computing systems are considered as a sub-class of autonomic computing systems (Dagstuhl Seminar, 2015). In analogy to an autonomic computing system with its four self-chop properties (cf. Section 2.1.1), a computing system is considered to be *self-aware* if it possesses, and/or is able to acquire at run-time, the following three properties, ideally to an increasing degree the longer the system is in operation:

- *Self-Reflective*: The system is aware of its software architecture, execution environment, and hardware infrastructure on which it is running as well as of its operational goals (e.g., performance and resource efficiency targets, or other QoS requirements)
- *Self-Predictive*: The system is able to predict the effect of dynamic changes (e.g., changing service workloads) as well as predict the effect of possible adaptation actions (e.g., changing system configuration, adding/removing resources),
- *Self-Adaptive*: The system proactively adapts as the environment evolves in order to ensure that its operational goals are continuously met.

The three properties in the above definition are obviously not binary, and different systems may satisfy them to a different degree. However, in order to speak of "self-awareness", all three properties must apply to the considered system.

To realize self-aware computing systems as defined above, novel model-based methods to design and engineer self-aware systems from the ground up are needed (cf. Chapter 1). The envisioned self-aware computing systems should have built-in self-reflective and self-predictive capabilities, encapsulated in the form of online system architecture models. In analogy to Descartes' dualism principle, these models are intended to serve as a *mind* to the system (the *body*) with a bidirectional link between the two. The models should capture the relevant influences (with respect to the system's operational goals) of the system's software architecture, its configuration, its usage profile, and its execution environment (e.g., physical hardware, virtualization, and middleware). The models are also assumed to explicitly capture the system's operational goals and policies (e.g., QoS requirements, service level agreements, efficiency targets) as well as the system's adaptation space, adaptation strategies and processes.

In general, self-awareness, as defined above, can be considered with respect to one or multiple QoS properties. The approach presented in this thesis is focused on managing system performance requirements and resource efficiency. Note that previously in Section 2.1.1, definitions of the various self-* properties were presented, among them also a definition of the "self-aware" property. Compared to the definition given in Section 2.1.1, the definition given here is more detailed and explicitly includes the self-reflection property. In the context of self-aware computing systems, we are thus looking for sophisticated modeling and prediction techniques that can be leveraged to achieve a detailed reflection of the system, which will be discussed in detail in Chapter 4.

2.2. Software Performance Engineering

Software engineering aims at addressing the challenges of the software development process by means of an engineering discipline. A characteristic of such a discipline is the availability of a catalog of methods and practices plus guidelines for the systematic selection of these practices. Software Performance Engineering (SPE) is a systematic quantitative approach to the cost-effective development of software systems to meet performance requirements (Smith, 1981). Motivated by the fact that systems suffering from insufficient performance can cause projects to fail, a major goal of SPE is to conduct performance evaluation of software architectures as early as possible (Smith and Williams, 2002). To achieve this goal, predicting the performance of software systems is an important step, which is in the focus of research since the end of the 1990's (Koziolek, 2010). Today, performance models and performance prediction techniques are gaining importance in research on cloud computing and Green IT, as such methods and techniques promise the ability to predict at run-time how the performance of running applications would be affected if service workloads change, or to predict the effect of changing resource allocations. We refer to this ability as *online performance prediction*.

In the following, we present the terminology for the two types of performance models we use in this thesis (Section 2.2.1). Next, Section 2.2.2 introduces the concepts of model-driven performance engineering and in Section 2.2.3, we give an overview of Palladio Component Model (PCM) as example of an architecture-level performance model. Finally, Section 2.2.4 explains how architecture-level performance models can be used for online performance prediction.

2.2.1. Performance Models: Classification and Terminology

We distinguish between predictive performance models and descriptive architecture-level performance models. *Predictive* performance models capture the temporal system behavior and can be used for performance prediction by means of analytical or simulation techniques (e.g., queueing networks). 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. 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. Finally, predictive performance models typically impose many restrictive assumptions such as single workload class, single-threaded components, homogeneous servers, or exponential service and request inter-arrival times.

Descriptive architecture-level performance models provide means to model the performance-relevant aspects of system architectures at a more detailed level of abstraction. Architecture-level in this context is meant in a broader sense covering both the system's software architecture and its execution environment. Such models describe the software architecture using models such as UML that are annotated with descriptions of the system's performance-relevant behavior. Over the past decade, a number of architecture-level performance meta-models have been developed by the performance engineering community which we review later as part of the related work presented in Section 3.2.2. The common goal of these efforts is to make it possible to predict the system performance for a given workload and configuration scenario by transforming architecture-level performance models into predictive performance models in an automatic or semi-automatic manner (cf. Section 2.2.2).

Architecture-level performance models provide a powerful tool for performance prediction. They are normally built manually during system development and are intended for use in an offline setting at design and deployment time to evaluate alternative system designs and/or to 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 vs. run-time (Brosig et al., 2013). This specific challenge is addressed in the work of (Brosig, 2014), which is focused on using architecture-level performance models for performance prediction at run-time.

2.2.2. Model-Driven Software Performance Engineering

A starting point for most approaches to SPE is to describe the architecture of the software system (Balsamo et al., 2004; Koziolek, 2010). The architectural model can then be annotated with estimated or measured information about the system performance (cf. Figure 2.6). Next, the architecture-level performance model is transformed into a predictive performance model (e.g., a queueing network or queueing Petri net) to predict and analyze the metrics of interest.

In the area of SPE, the idea of using model-driven techniques gained attention because such techniques provide means to formalize the syntax and semantics of performance models. Furthermore, automated transformation from source models (e.g., an architecture-level model) to target models (e.g., predictive performance models) can be performed. The automatic transformation and processing can thus simplify the software performance engineering process and make it less error-prone. In the following section, we present an example of a model-driven approach to describe and analyze the performance behavior of component-based software systems.

2.2.3. The Palladio Component Model (PCM)

The Palladio Component Model (PCM) is a domain specific modeling language to describe the performance-relevant aspects of component-based software architectures as often used in business information systems. It is designed to enable early performance, reliability, and cost predictions for software systems and is aligned with a component-based software development process. In addition to providing a meta-model, there is a tool to create

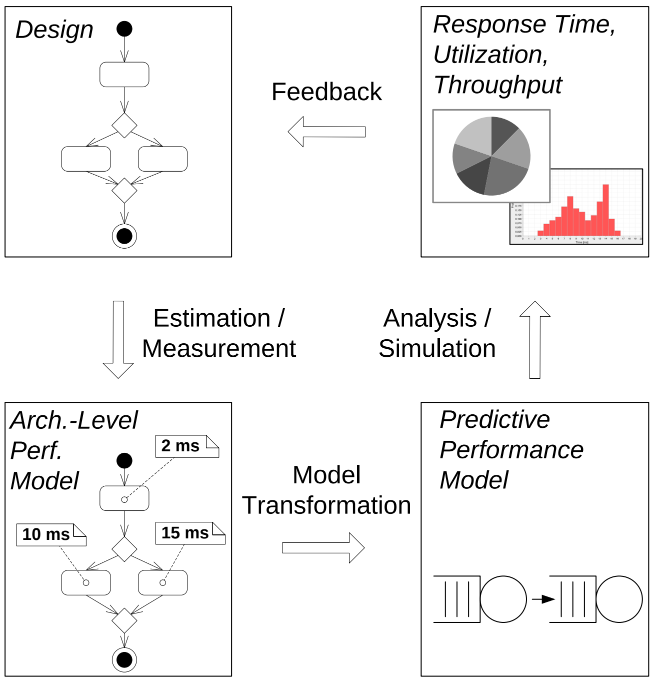


Figure 2.6.: Model-driven software performance engineering (cf. Huber et al., 2010a).

and analyze PCM model instances, the PCM Bench (2014). The following gives a brief overview of the concepts of PCM. A more detailed and technical description is given by Becker et al. (2009).

In PCM, software components are units of composition with explicitly defined provided and required interfaces (Szyperski et al., 2002). The performance of such a software component is influenced by four factors (Koziolek, 2010), depicted in Figure 2.7. The PCM meta-model provides means to take all of these factors into account.

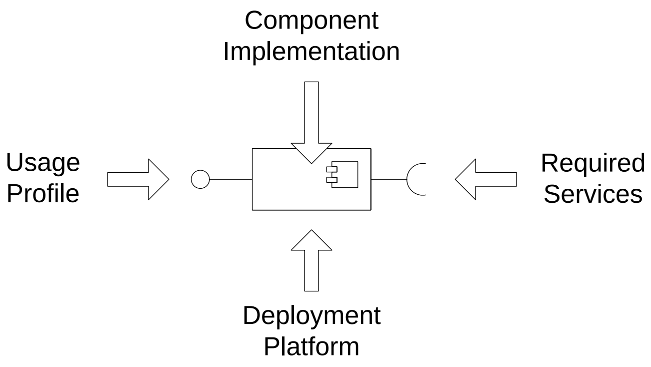


Figure 2.7.: Factors influencing component performance (cf. Koziolek, 2010).

A PCM model instance can be divided into four different views according to these four influence factors. Each influence factor is modeled by a specific role, specifying the performance-influencing factors in separate models.

The *Component Developer* models the components and their implementation. At first, the interfaces (comparable to signature lists) that are provided or required by a component are specified. Provided interfaces specify the services a component offers. Required interfaces are external services that the component needs to fulfill its purpose. Furthermore, the component developer models the internal behavior of the provided service(s). To this end, the PCM provides a description language, called Resource Demanding Service Effect Specification (RD-SEFF), to specify the control flow (e.g., branches, loops, external service calls) and the resource usage of the provided service(s).

The *System Architect* models the structure of the system. By interconnecting components via their provided and required interfaces, it can be specified which specific services a component uses. Hence, the overall system's performance depends on the selection of components, e.g., if a database cache component is used or not.

The *System Deployer* maps the components of the system model to physical resources. Thereby, the influence of the deployment platform on the system performance is specified. Therefore, he must model the hardware environment the system is executed on (like processor speed, network links, memory usage, etc.). Furthermore, he deploys the system components on the specified hardware resources.

The *Domain Expert* models the usage profile of the whole system. He describes which parts of the system are used by the system's end-users. This description also comprises the type of workload issued to the system (open or closed). The workload specifies for example how many users invoke the system or the interarrival time of invocations.

Usually the performance of a software system does not depend on constant parameter values because, e.g., resource demands or system usage may vary during execution. PCM offers *random variables* to express the uncertainty of such parameters. Random variables can be specified by various probability distribution functions. Furthermore, it is possible to specify mathematical expressions by combining variables with mathematical operators.

2.2.4. Online Performance Prediction

A central element of the approach presented in this thesis is the use of an architecture-level performance model to predict the impact of adaptation actions by adapting and analyzing the model on-the-fly. Since this analysis has to be performed at run-time, where only limited time and restricted monitoring data may be available, Brosig (2014) has presented an approach for tailored *online performance prediction*. In contrast to black-box approaches to performance prediction at run-time, such as (Menasce and Virgilio, 2000; Gambi et al., 2013), the techniques of Brosig allow us to vary and analyze the impact of multiple degrees of freedom such as system configurations, service compositions, and resource allocations. These prediction techniques are based on the Descartes Modeling Language (DML), a novel architecture-level modeling formalism specifically designed for performance and resource management in online scenarios. As major parts of DML are part of the contribution of this thesis, DML will be presented in Section 4.2.

The online performance prediction techniques are able to answer *performance queries* that can be derived from questions such as: What performance would a new service deployed on the infrastructure exhibit? How much resources should be allocated to it? How should the workloads of the services be partitioned among the available resources? If any service experiences a load spike or a change of its workload, 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? When answering such queries at run-time, there is a trade-off between prediction accuracy and time-to-result. There are situations where the prediction results need to be available in a short period of time such that the system can be adapted *before* SLAs are violated. Accurate fine-grained performance prediction comes at the cost of a higher prediction overhead and longer time-to-result, whereas coarse-grained performance predictions allow speeding up the prediction process. The challenge is to balance the trade-off between prediction accuracy and prediction speed.

Performance Queries

The approach presented by Brosig (2014) allows to conduct performance predictions on-the-fly where each prediction is tailored to answering a given performance query. A performance query describes which specific performance metrics of which entities are of interest. For instance, when triggering a performance prediction, the 90th percentile response time of a specific service or the utilization of a specific resource, such as a database server, may be of interest. For situations where the prediction speed is critical, performance queries provide the option to speed up the prediction process at the expense of prediction accuracy. Note that in this context, *prediction accuracy* refers to the accuracy of the model solving approach and *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.

Therefore, we use trade-off weights that are ranked in an ordinal scale, defined as 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 $w \in W$. Weight $w_\perp$ has the semantics of highest prediction accuracy compared to the other $w \in W$. A trade-off specification is then given by a selected weight $dw \in W$ that is chosen by the issuer of the performance query. To sum up, the prediction process is tailored to the required performance metrics as well as to a given trade-off weight between prediction accuracy and speed.

```
SELECT r.utilization, s.avgResponseTime
CONSTRAINED AS 'FastResponse'
FOR RESOURCE 'AppServerCPU1' AS r,
    SERVICE 'newOrder' AS s
USING dmmConnector@modelLocation;
```

Listing 2.1: Constrained Query

The notion of a performance query, formalized in Gorsler et al. (2014), provides a declarative interface to performance prediction techniques to simplify the process of using architecture-level performance models for performance analysis. The query language provides a notation to express the required performance metrics for prediction as well as the goals and constraints of a specific prediction scenario. An illustrative example of such a performance query is depicted in Listing 2.1. It queries for the average response time of a 'newOrder' service as well as the average utilization of an application server CPU, and requests a 'FastResponse' prediction (equivalent to $w_\top$).

Tailored Prediction Process

Figure 2.8 provides an overview of the prediction process showing the individual steps and their inputs and outputs. The prediction process is triggered by a performance query referring to a DML instance specified in the `USING` clause of the query.

The model composition step marks those parts of DML instance 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 behavior descriptions such as a fine-grained behavior, a coarse-grained behavior, and a black-box description, the model composition step chooses a behavior description that provides adequate means to predict the requested performance metrics considering the specified trade-off weight.

The next step traverses DML instance starting with the usage scenarios specified as part of the usage profile model. First, it resolves the probabilistic characterizations of the parameter dependencies of DML's application architecture meta-model. Second, it parameterizes

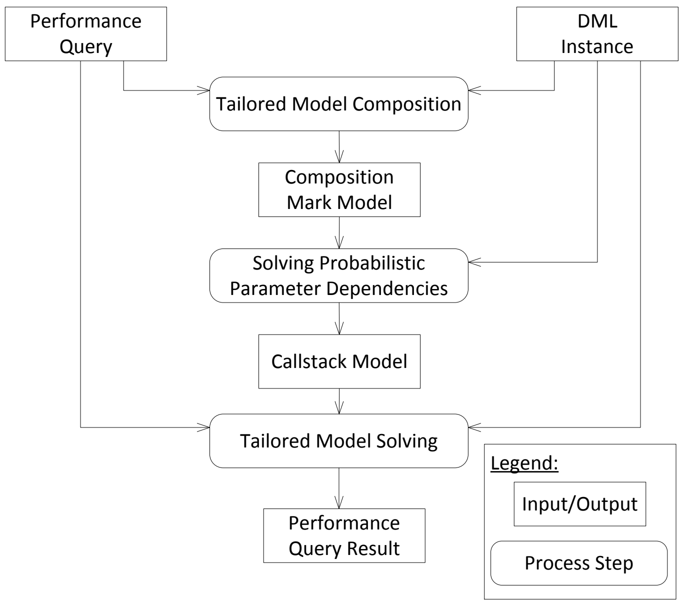


Figure 2.8.: Overview of the online performance prediction process (cf. Brosig, 2014).

the performance model on-the-fly using monitoring data. The output is a call graph together with the corresponding model parameter values, denoted as *callstack model*. The call graph determines how the performance model has to be traversed for the performance prediction.

The next step is the tailored model solving, i.e., it predicts the requested metrics considering the given trade-off weight. 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 the specific predictable metrics, the prediction accuracy, and the prediction speed.

- 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 synchronous resource possession (Menasce and Virgilio, 2000; Li et al., 2009; Gilmore et al., 2005).

As analytical solving technique, the approach for tailored online performance prediction developed by Brosig (2014) applies, among others, asymptotic bounds analysis (Bolch et al., 1998). As simulation technique, it transforms a DML instance to a Queueing Petri Net (QPN) (Kounev, 2006) and simulates it using the SimQPN simulation engine (Spinner et al., 2012). The overhead of a bounds analysis is lower than the overhead of a transformation to a QPN. A bounds analysis can quickly provide upper 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). Both the transformation itself and the simulation are tailored to the given performance query. Depending on whether average response times or response time distributions are requested, different places and different token colors are tracked as they traverse the simulated QPN. The tracking is implemented using the *Probes* feature provided by SimQPN (Spinner et al., 2012). Moreover, SimQPN supports fine-grained options to control what type and amount of data is logged during the simulation run. Finally, SimQPN's simulation stopping criterion is tailored to use non-overlapping batch means for estimating the variance of mean residence times configured to stop when a certain confidence level has been reached. The significance level and the desired width of the provided confidence interval are configured in a way to reflect the specified trade-off weight.

3. Related Work

In recent years, many approaches for automated management of performance and resource efficiency as well as other system QoS properties have been proposed in the literature. The related work considered in the following overview resides on the intersection of the areas of autonomic computing, software engineering, and performance engineering (cf. Figure 2.1). On the one hand, there is related work in the areas of autonomic computing and software engineering of self-adaptive systems. Related approaches in these areas typically start at a high level of abstraction, the system architecture, to achieve self-adaptation and QoS management. Recent articles surveying approaches in the area of autonomic computing and self-adaptive software have been presented by Huebscher and McCann (2008) and Salehie and Tahvildari (2009), respectively. On the other hand, there is related work in the area of (model-driven) performance engineering, focused on managing the performance of IT systems at design-time and run-time. Such approaches normally start from different types of performance models and use these models for evaluating the performance of design alternatives or for capacity planning at system run-time. In this line of research, Becker et al. (2012) survey approaches on model-driven performance engineering of self-adaptive systems. Furthermore, Balsamo et al. (2004) survey different types of performance modeling formalisms, and Koziolek (2010) presents a survey on component-based performance modeling approaches.

In this chapter, we present an overview of related approaches from the above-mentioned two major groups. First, we focus on related work from the area of architecture-based self-adaptive software systems and (model-driven) engineering of such systems (Section 3.1). We review approaches that employ some kind of (architecture-level) modeling formalism or model-driven technique to reason about different QoS properties of the system and its adaptation behavior. For each approach, we summarize its core idea and discuss its restrictions and limitations compared to the approach presented in this thesis. Second, we discuss approaches focused on model-based performance and resource management at run-time using different types of performance models (Section 3.2). We differentiate the considered approaches according to the employed modeling formalism, explicitly distinguishing between predictive performance models and descriptive (architecture-level) performance models. Furthermore, we discuss the suitability of the employed modeling formalisms for online performance prediction as well as their capabilities for automated decision-making and system adaptation at run-time. Finally, we give a brief overview of approaches that focus on modeling formalisms to describe the adaptation behavior of autonomic and self-adaptive systems (Section 3.3).

3.1. Architecture-Based Self-Adaptive Software Systems

Over the last 15 years, many approaches for engineering of self-adaptive software have been proposed in the autonomic computing and software engineering communities (cf. Huebscher and McCann, 2008; Salehie and Tahvildari, 2009). In this section, we discuss approaches that employ some kind of modeling formalism to describe the architecture of the system to be adapted or its adaptation behavior. In the considered approaches, the models created with such formalisms are then either used i) to support different kinds of reasoning during system adaptation, ii) for the engineering of a self-adaptive system by generating software artifacts, or iii) by system architects to design self-adaptive systems or to developed adaptation processes. In general, such approaches are independent of specific QoS properties but have in common that they start at the architecture level. We separate the considered approaches into two groups: generic methods for engineering and evaluating self-adaptive software systems (Section 3.1.1) and specific architecture-based self-adaptation approaches (Section 3.1.2).

3.1.1. Engineering and Evaluation of Self-Adaptive Software Systems

EUREMA

ExecUtable RuntimE MegAmodels (EUREMA) (Vogel and Giese, 2014) is a model-driven engineering approach that enables the specification and execution of adaptation engines for self-adaptive software with multiple feedback loops. EUREMA provides a domain-specific modeling language to specify adaptation engines using two types of diagrams. To model a feedback loop with its adaptation activities and runtime models, the EUREMA language provides a behavioral feedback loop diagram (FLD). Such a diagram specifies a feedback loop with its adaptation activities and the used run-time models-. In this context, run-time models provide up-to-date and exact information about the system (cf. Blair et al., 2009). A structural layer diagram (LD) describes how the described feedback loop and the adaptable software system are related to each other in a specific situation of the self-adaptive software. Thus, an LD provides an architectural view that considers feedback loops encapsulated as black-box modules while white-box views are provided by FLDs. Furthermore, Vogel and Giese (2014) provide a runtime interpreter that supports the execution of the specified feedback loops. In contrast to existing work on self-adaptive software, EUREMA covers the specification and the execution of adaptation engines. The EUREMA language supports the explicit modeling of feedback loops and their coordinated execution.

EUREMA provides means to support the explicit design, execution, and adaptation of feedback loops at a high level of abstraction. Internally, EUREMA relies on runtime models (Blair et al., 2009) to describe the adaptable software and its environment, which are updated by monitoring the software and its environment. However, the authors do not elaborate in detail how these runtime models are used within the EUREMA runtime interpreter to predict the impact of adaptation actions and how such predictions could be used to manage performance and resource efficiency in software systems. To evaluate the expressiveness of the approach, the authors have applied EUREMA to different implementations of self-adaptive systems (PLASMA, Rainbow, and DiVA, cf. Section 3.1.2). However, the results provide no insights about the question if and how EUREMA can be used for autonomic performance-aware resource management at run-time.

FORMS

Weyns et al. (2012) claim that models and frameworks like Archstudio, Rainbow, or MUSIC (cf. Section 3.1.2) have achieved noteworthy success in many domains, but they are not formal enough to unambiguously describe and reason about the primary architectural characteristics of self-adaptive systems. Thus, they present a FOrmal Reference Model for Self-adaptation (FORMS) that can be used to describe the crucial aspects of distributed self-adaptive software systems relevant for reasoning about how the system adapts itself or how the system coordinates monitoring and adaptation in a distributed setting. To this end, FORMS provides a small number of formally specified modeling elements that correspond to the key concerns in the design of self-adaptive software systems.

Although FORMS supports engineers in describing the key concerns of their architectures and reasoning about important properties via supporting tools, it does not provide means to actually implement a self-adaptive system. Thus, it cannot be directly used to achieve autonomic performance-aware resource management, but it provides valuable concepts for reasoning about such adaptation techniques.

DYNAMICO

Another approach to support the engineering of self-adaptive systems is DYNAMICO (Villegas et al., 2013). The DYNAMICO (Dynamic Adaptive, Monitoring and Control Objectives) model is a reference model that supports the designers of self-adaptive systems in assuring the coherence of adaptation mechanisms, adaptation goals, and monitoring mechanisms with respect to changes in both adaptation goals and adaptation mechanisms. DYNAMICO proposes three different feedback loops for three different control elements: a control objectives manager that manages changes in the adaptation goals, an adaptation controller mechanism that handles changes directly at the target system level, and a monitoring infrastructure controller mechanism that manages changes that require the deployment of different or additional monitoring infrastructures. Thereby, DYNAMICO supports three different types of adaptation—preventive, corrective and predictive adaptations—depending on the different interactions implemented in the control elements. As a result, DYNAMICO emphasizes the visibility of these control elements and constitutes a guide to design self-adaptive systems in which the system goals, the target system itself, or the monitoring infrastructure must be adapted.

The approach presented by Villegas et al. (2013) contains valuable concepts and ideas for engineering self-adaptive systems and for the approach presented in this thesis. More specifically, adapting monitoring strategies to changes in the adaptation goals or user requirements are interesting for deriving performance models of the system. However, they do not elaborate in more details how their approach can be used to leverage performance models for predicting the impact of changes in the system environment and to consider the predicted impact to guide system adaptation.

SimuLizar

With SimuLizar, Becker et al. (2013) propose an approach to support the engineering of the self-adaptation logic of self-adaptive software systems. SimuLizar is based on the Palladio Component Model (PCM) (cf. Section 2.2.3) and provides modeling support for self-adaptation rules as well as a simulation engine that enables the performance prediction of self-adaptive systems and their various configurations. Thereby, SimuLizar helps to analyze and validate requirements of the transient phases of self-adaptive software systems at design-time.

SimuLizar inherits the modeling abstractions for describing the software system architecture and performance properties from PCM. Thus, like PCM for component-based software systems, SimuLizar is focused on modeling and evaluating self-adaptive software systems at design-time. However, these modeling abstractions are of limited expressiveness when used at run-time (cf. Brosig, 2014). For example, at run-time, explicit support for describing the performance behavior of the software system at multiple abstraction levels is necessary since one 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. Moreover, since SimuLizar is focused on evaluating the self-adaptation logic of the system at design-time, it is not intended to actually adapt the system to changes in the system environment.

3.1.2. Architecture-Based Self-Adaptation Approaches

Archstudio

Oreizy et al. (1999) present an infrastructure for system evolution and system adaptation. To provide architecture-based adaptation, their infrastructure relies on the explicit representation of software components, their interdependencies, and their environmental assumptions. In their approach, the software system's architecture is described as a dynamic architecture, characterized using graphs of components and connectors, and architectural changes are regarded as graph-rewrite operations. The approach relies on C2 (Taylor et al., 1996) and Weaves (Gorlick and Razouk, 1991) as architectural formalisms to describe the software dynamics at the architecture level, and an architecture evolution manager for guiding and checking all modifications directed toward the architectural model.

Although the considered adaptations encompass a broad spectrum, such as replacement of isolated components as well as reconfigurations that are pervasive and physically distributed, this approach is limited on adapting the software architecture to react on changes in the system's operational environment and of the software system's goal. Furthermore, the architectural models and the encoded dynamics are used to reason about where and how to reconfigure the system, but they do not provide means to reason about the adaptation impact, e.g., on the performance behavior of the system. Although the authors present valuable concepts important for engineering self-adaptive software systems, this approach resides at a high level of abstraction (Oreizy et al., 1999). It remains unclear how the presented concepts are integrated to support self-adaptive software for performance and resource management. Furthermore, the approach does not provide means to describe system adaptation processes at the model-level.

Darwin

Darwin is an Architecture Description Language (ADL) to structure distributed systems consisting of multiple concurrently executing and interacting components (Magee et al., 1995). Thereby, Darwin allows the construction of systems from a hierarchically structured specification of the set of used component instances and their interconnection. Sykes et al. (2008) use Darwin as the basis for a three layer model to derive adaptation plans and adapt the component model accordingly. On top, the goal management layer generates reactive adaptation plans from high-level goals. The change management layer in the middle uses the generated plans to construct component configurations and directs this to the bottom layer, the component layer.

However, this approach is restricted to adapting the component architecture of the system and does not consider adapting the resource infrastructure of the system. Furthermore, Darwin is not intended to capture the performance-relevant properties or behavior of the software components and it is thus unsuitable for performance predictions and reasoning about the impact of adaptation actions on the system performance and resource efficiency.

Rainbow

Rainbow (Garlan et al., 2004) is a system that provides an engineering approach and a framework of mechanisms to monitor a system and its execution environment, reflect observations into an architectural model of the system, determine problem states, select a course of action, and effect changes (Garlan et al., 2009). Rainbow uses an architectural model together with existing architectural analysis techniques to reason about changes that should be made to a system to improve the system's achievement of the target quality attributes. Like Archstudio (Oreizy et al., 1999), Rainbow uses ADLs as modeling formalism in which the system architecture is represented as a graph of interacting components. In addition, Rainbow provides Stitch (Cheng and Garlan, 2012), a programming-language-like notation to express repair strategies in a form that can be analyzed and automated by the Rainbow framework.

Like in Archstudio, the employed architectural model is generic and does not explicitly model performance-related properties that can be leveraged at run-time to reason about the performance impact of changes in the system environment or the impact of adaptation actions. The limitation of Stitch is that adaptation strategies are expressed in a strictly deterministic process-oriented fashion where each step evaluates a set of condition-action pairs to select the adaptation action. This limits the flexibility to specify adaptation processes to adapt the system in situations of uncertainty.

DiVA

The approach developed in the DiVA project (Dynamic Variability in complex, Adaptive systems) is a model-based approach for specifying dynamically adaptive software systems based on software product lines (Morin et al., 2009). The approach employs four meta-models providing means to model a system's software architecture, execution environment, and variability, as well as a reasoning model that can be used to describe how different features of the software impact the system QoS. To adapt and optimize component-based applications at run-time, DiVA uses several formalisms such as event-condition-action rules or goal-based optimization rules.

In contrast to our approach, DiVA is focused on automating the dynamic adaptation process in software product lines to adapt component-based applications at run-time to evolving design specifications. Thus, the expressiveness of the models to specify performance-relevant aspects of the software components and the system environment is limited and the adaptation possibilities of DiVA are restricted to the software architecture level.

Genie

Another approach based on a domain-specific modeling language for modeling adaptive systems is Genie (Bencomo et al., 2008; Bencomo and Blair, 2009). The modeling language used in this approach allows to capture the dynamic aspects of component frameworks such as structural variability as well as environment and context variability (changes in the environment or requirements of the system). To describe self-adaptation as transitions between configurations of the adaptable software, Genie uses policies of the form of "on-event-do-actions" applying architectural changes with the goal to improve the current state of the system with respect to given QoS properties. Using generative techniques, Genie allows the systematic construction of middleware-related software artifacts from high level description models. With Genie, new reconfiguration policies can be modeled and generated offline while the system is running.

However, Genie does not support reasoning about the performance-related aspects of a software system and it relies on specific middleware and component frameworks. Furthermore, like other approaches presented in this section, Genie is focused on the middleware and component-based applications and thus, the approach does not support the adaptation of the infrastructure of modern IT systems and services.

MADAM

Another model-driven engineering approach for the development and operation of context-aware, self-adaptive applications is MADAM (Floch et al., 2006; Geihs et al., 2009). The MADAM framework consists of a middleware that supports the dynamic adaptation of component-based applications using an application architecture model based on the Unified Modeling Language (UML), in which components can be annotated with QoS properties for adaptation reasoning. In this approach, the application developer can describe the application variability using a platform-independent adaptation model which is then transformed by corresponding tools into a representation that is used by the middleware for adaptation management. The adaptations supported by MADAM are parameter adaptation, i.e., a fine tuning of applications through the modification of program variables and deployment parameters, and compositional adaptation, i.e., the modification of the application component structure and the replacement of components.

The modeling formalism used in MADAM resides at a high level. Components are simply annotated with properties to qualify the services that are provided or required by components, which limits the possibilities for fine-grained performance analysis. Furthermore, MADAM does not support adaptations of the infrastructure such as migrating VMs or changing VM parameters. Finally, this approach is focused on applications in mobile computing scenarios and the developed applications are targeted to react dynamically on fluctuations in network connectivity, battery capacity, appearance of new devices and services, or to changes of user preferences (Geihs et al., 2009).

MUSIC

This approach is the successor of MADAM. While MADAM is focused on mobile computing scenarios with the underlying assumption of a closed world computing environment, MUSIC targets ubiquitous computing environments that are characterized by openness, heterogeneity, and dynamic service discovery and binding (Hallsteinsen et al., 2012).

According to Hallsteinsen et al. (2012), the main goal of MUSIC is to simplify the development of adaptive applications that will operate in open and dynamic ubiquitous computing environments and adapt seamlessly and without user intervention in reaction to context changes. As adaptation possibilities, MUSIC supports setting configuration and application parameters, replacing components and service bindings, and redeploying components on the distributed computing infrastructure. In its core, MUSIC implements a MAPE-K control loop, employing a coarse-grained QoS-aware architecture model as shared knowledge to realize the dynamic and automatic adaptation of applications and services. MUSIC's adaptation-planning uses utility functions to evaluate the utility of alternative application configurations in response to context changes to select a feasible configuration for the current context, and to adapt the application accordingly.

However, like MADAM, the capabilities of this approach to manage performance and resource efficiency in an autonomic manner are limited as it still uses too coarse-grained modeling abstractions of the performance properties. Although the application scope of MUSIC has been extended compared to MADAM, the adaptation capabilities of the approach remain at the application level and do not consider changes at the infrastructure level.

PLASMA

Tajalli et al. (2010) present an approach to self-adaptive software that utilizes modeling and planning techniques, called Plan-based Layered Architecture for Software Model-driven Adaptation (PLASMA). The approach uses models based on ADLs and a specification of the system's goals as inputs to generate adaptation plans in response to changing system requirements and goals. PLASMA employs three adaptive layers: i) the bottom application layer describes the application-level software components, ii) the middle layer (called the adaptation layer) monitors, manages, and adapts components in the application layer, and iii) on top the planning layer that manages the adaptation layer and the generation of plans based on goals and component specifications provided by the system architect. In PLASMA, it is the responsibility of an adaptation planner to find an adaptation plan that transforms the current architecture of the application layer to the application goal. The authors evaluate their approach in the context of a robotic application in which three or more robots that form a convoy follow a leader robot.

Like the previously described approaches, PLASMA is restricted to adaptations at the application architecture level. Although the presented approach is an interesting solution to generate plans for adapting the software to changing system requirements, the adaptation planning currently provides no capabilities to predict the impact of possible adaptations on the system performance and consider this during adaptation planning.

GRAF

The Graph-based Runtime Adaptation Framework (GRAF) presented by Amoui et al. (2012) is another model-centric approach that uses graph-based models that are interpreted at run-time to manage system adaptation. In this approach, an adaptation manager controls the adaptable software by manipulating a runtime model instead of directly operating on the adaptable software. The runtime model corresponds to a graph that describes the adaptable software's state and a collection of behavior descriptions, captured using a subset of UML activity diagrams. The adaptation behavior is specified as a set of adaptation rules, whereas a single adaptation rule corresponds to an Event–Condition–Action (ECA) rule (Widom and Ceri, 1996).

GRAF is restricted to adaptations at the architecture-level of software systems, relying on parameter adaptation and compositional adaptations. Furthermore, the modeling formalism used in GRAF provides only limited expressiveness for performance-related properties and for specifying adaptation processes.

SLAstic

SLAstic is an online capacity management approach for component-based software systems with the goal to reduce the operating costs of software systems (van Hoorn et al., 2009; van Hoorn, 2014). Resource efficiency, in terms of the number of allocated data center resources, is improved by executing architecture-level adaptations at run-time based on current workload situations. Based on monitoring and predicting the current and future performance and efficiency measures, SLAstic plans and executes system adaptations. Monitoring data is obtained with Kieker (van Hoorn et al., 2012), a framework that enables application performance monitoring and architecture discovery, and performance metrics are predicted using simulation (von Massow et al., 2011).

Currently, SLAstic contributes to the planning and execution phases of the MAPE-K control loop (cf. Section 2.1.2). However, SLAstic's adaptation possibilities are focused mainly on the software architecture, e.g., de-/replicating or migrating software components. Thus, it desists form a detailed modeling of the resource landscape in which the software system is deployed. Furthermore, SLAstic provides no means for the explicit specification of adaptation actions and processes that can be performed to adapt the software system.

3.2. Model-Based Performance and Resource Management

In this section, we give an overview of different approaches to performance and resource management at design-time and run-time. More specifically, we discuss approaches that use some kind of performance model to describe the performance behavior of the system and predict the performance impact of changes to the system architecture or changes in the system environment. These performance predictions are then used to, e.g., evaluate design decisions or for capacity planning. In the following, we further separate the considered approaches into two groups depending on the applied modeling formalism. The first group of related work applies predictive stochastic performance models to predict the performance impact of changes (Section 3.2.1), whereas the second group relies on descriptive architecture-level performance models to describe the performance behavior at the architecture level and applies model transformation techniques for performance prediction (Section 3.2.2).

3.2.1. Predictive Performance Models

With the adoption of virtualization and cloud computing, many approaches to online performance and resource management in dynamic environments have been developed in the research community. Such approaches are typically based on control theory feedback loops, machine learning techniques, or stochastic performance models, such as layered queueing networks or stochastic Petri nets. Approaches based on feedback loops and control theory (e.g., Abdelzaher et al., 2002; Almeida et al., 2010) can normally guarantee system stability by capturing the transient system behavior (Almeida et al., 2010). Machine learning techniques 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; Kephart et al., 2007). Performance models are typically used in the context of utility-based optimization techniques. They are embedded within optimization frameworks aiming at optimizing multiple criteria such as different QoS metrics (Verma et al., 2008; Jung et al., 2010; Mi et al., 2010). Existing work in this area mainly uses predictive performance models that capture the temporal system behavior where the platform is normally abstracted as a *black-box*, that is, the software architecture and configuration are not modeled explicitly (e.g., Chen et al., 2005; Bennani and Menasce, 2005; Zhang et al., 2007; Urgaonkar et al., 2005; Jung et al., 2008). Such models include queueing networks (e.g., Menasce and Virgilio, 2000), layered queueing networks (e.g., Li et al., 2009), queueing Petri nets (e.g., Kounev, 2006), stochastic process algebras (e.g., Gilmore et al., 2005), statistical regression models (e.g., Eskenazi et al., 2004), Kriging models (e.g., Gambi et al., 2013), fuzzy logic (e.g., Jamshidi et al., 2014), or control engineering approaches based on multiple models (e.g., Patikirikorala et al., 2012). Models are typically solved analytically, e.g., based on mean-value analysis, as presented by Zhang et al. (2007), or by simulation where analytical solution is not a viable option (Jung et al., 2008).

In summary, existing model-based techniques for online performance and resource management typically abstract the system as a black-box and do not explicitly model the software architecture and execution environment. For example, they do not distinguish performance-relevant behavior at the virtualization level vs. at the level of applications hosted inside the running VMs. However, explicitly considering such aspects is crucial for online performance prediction and model-based system adaptation at run-time, as black-box prediction approaches do not provide enough flexibility to analyze the impact of fine-granular adaptation actions.

3.2.2. Architecture-Level Performance Models

In the software performance engineering community, a number of modeling approaches for building architecture-level performance models of software systems have been proposed over the last decade (cf. Koziolek, 2010; Balsamo et al., 2004). Such models provide modeling constructs to capture the performance-relevant behavior of a system's software architecture as well as some aspects of its execution environment. The most prominent meta-models are the UML SPTP (Object Management Group (OMG), 2005) and UML MARTE profiles (Object Management Group (OMG), 2011c), both of which are extensions of UML as the de facto standard modeling language for software architectures. Further proposed meta-models include SPE-MM (Smith et al., 2005), CSM (Petriu and Woodside, 2007), KLAPER (Grassi et al., 2007) and PCM (Becker et al., 2009). The common goal of these approaches is to predict the modeled system's performance behavior by transforming the architecture-level performance model into a predictive performance model.

The restriction of the presented approaches is that they are designed for use at system design-time in an offline setting. Therefore, they typically assume a static system architecture and execution environment. However, explicitly considering dynamic changes and being able to predict their effect at run-time is indispensable for ensuring predictable performance and automated decision-making. A recent survey on model-based approaches to engineering of software systems confirms the importance of considering the dynamic aspects of modern IT systems and services as part of architectural models (Becker et al., 2012). However, the survey also shows that currently, there are no performance modeling approaches that support the explicit modeling of adaptation strategies and processes at the architecture level. Note that at the time of publication of this survey, the adaptation process modeling language presented in Chapter 6 as integral part of DML was under development.

3.3. Adaptation Process Modeling Languages

In the field of software engineering, there exist various domain-specific languages and modeling formalisms to describe the dynamic behavior of software. For example, UML provides activity diagrams to describe the overall control flow of a software system. This section gives a brief overview of approaches that aim at modeling the adaptation behavior of autonomic and self-adaptive systems at the architecture level.

Cheng and Garlan (2012) introduce Stitch, a programming language-like notation to express repair strategies in a form that can be analyzed and automated by the Rainbow framework (cf. Section 3.1). In Stitch, adaptation strategies as decision trees are constructed from adaptation tactics, which are in turn defined in terms of more primitive operators (Cheng and Garlan, 2012). However, strategies refer to tactics in a strictly deterministic, process-oriented fashion. Therefore, the knowledge about system adaptation specified with Stitch is still application-specific, making it difficult to adapt in situations of uncertainty.

Da Silva and de Lemos (2009) use Yet Another Workflow Language (YAWL) (van der Aalst and ter Hofstede, 2005), a modeling language based on Petri nets, to coordinate the complex adaptation process of a self-adaptive system. The authors execute architectural reconfiguration using dynamic workflows to adapt to changing requirements at run-time. They distinguish between abstract and concrete workflows, concepts which correspond to the concepts of strategies and tactics proposed by Cheng et al. (2006); Cheng and Garlan (2012). However, the focus of this approach is to automatically generate workflows for self-adaptation. It does not integrate a detailed architecture-level performance model to evaluate the impact of the adaptation actions.

Other domain-specific languages to describe adaptation behavior or processes can be found in the area of service-oriented systems and business process modeling. For example, Service Activity Schemas (SAS), introduced by Esfahani et al. (2009), is an activity-oriented language for modeling a software system's functional and QoS requirements. SAS is intended for user-driven composition and adaptation of service-oriented software systems. Another example is the Business Process Model and Notation (BPMN) (Object Management Group (OMG), 2011a), which is a graphical representation for specifying business processes in a business process model. However, these approaches are mainly focused on modeling business processes and are thus not suitable for modeling the full spectrum of self-adaptive mechanisms from conditional expressions to algorithms and heuristics.

When it comes to languages for specifying how to adapt model instances, well-known methods from the area of graph grammars exist. For example, Agrawal et al. (2003) present an approach on how to define model transformations based on UML. Another example of a graph grammar language are Story Diagrams (Fischer et al., 2000), which are also based on UML and Java. However, these approaches remain on the model level, i.e., they cannot be used to execute real system adaptations without additional tools.

The shortcoming of all mentioned approaches is that they do not integrate a specific performance modeling formalism and are thus unsuitable for reasoning about the performance impact of the adaptation actions. Nevertheless, Stitch and Story Diagrams provide useful concepts for describing system adaptation behavior which will be used for evaluation purposes (cf. Chapter 6).

3.4. Summary

In summary, current modeling formalisms for describing the performance-relevant properties and behavior of modern IT systems have two major limitations which render them unsuitable for autonomic performance-aware resource management at run-time. First, they do not provide sufficiently expressive modeling abstractions to capture the details of the distributed and layered resource infrastructures of modern IT systems such that online performance prediction and adaptation decision-making at run-time can be supported. Second, as such modeling formalisms are intended for use at design-time, the modeling abstractions do not support describing dynamic aspects of modern IT systems, such as a system's degrees of freedom for adaptation, or possible adaptation strategies and actions.

Current approaches for developing architecture-based self-adaptive systems usually focus on maintaining the functionality of the adapted system and are not intended for maintaining performance and resource efficiency requirements. If the adaptation approaches are concerned with the QoS properties of the system, such approaches usually aim at improving multiple QoS properties and therefore provide only coarse-grained modeling abstractions (like ADLs or UML-based methods) to describe the performance-behavior of the system. Thus, these approaches do not provide suitable online performance prediction capabilities necessary for performance-aware system adaptation. Furthermore, most of the presented approaches are based on component-based system architectures and focus only on adaptations at the application level. However, for performance and particularly for resource management, it is important that the resource environment of the entire system is considered explicitly as part of the designed system models and adaptation processes.

As a result, we conclude that existing model-based system adaptation approaches are not designed to provide both i) modeling abstractions to describe the system architecture and system adaptation processes and ii) modeling abstractions to describe the detailed performance-relevant properties and behavior of the system necessary for autonomic performance-aware resource management. Furthermore, current approaches generally pursue a reactive approach to system adaptation. Thus, there exists no holistic model-based system adaptation approach that covers all aspects relevant for proactive performance-aware resource management at run-time.

Part II.

Proactive Model-Based Performance and Resource Management

4. Proactive Model-Based System Adaptation

Many researchers agree that a promising approach to tackle the challenges and manage the complexity of autonomic and self-adaptive systems is the use of models (e.g., Blair et al., 2009; Cheng et al., 2009; Salehie and Tahvildari, 2009; de Lemos et al., 2011). In this chapter, we present a novel model-based approach for proactive performance and resource management focused on two major aspects. First, we aim at reducing complexity by abstracting from technical details and modeling the performance behavior of the system at the architecture-level to support autonomic decision-making at run-time (cf. Blair et al., 2009). Second, to provide a holistic model-based approach, we also model the system adaptation processes at the architecture-level. Thereby, we are able to leverage the benefits of model-driven engineering (MDE) techniques for the systematic engineering of self-adaptive software systems (cf. France and Rumpe, 2007; Kounev et al., 2010).

Core of our model-based adaptation approach is a refinement of a generic adaptation control loop concept used in the software engineering and autonomic computing community. The adaptation loop is based on the Descartes Modeling Language (DML), a novel domain-specific modeling language designed to capture both static and dynamic aspects of modern IT systems at the architecture-level. With DML, we can model all relevant influences of the system architecture and its resource landscape, as well as dynamic aspects like the system's configuration space and adaptation processes, in a generic, human-understandable and reusable way. We integrate DML in our adaptation loop and leverage its novel features and online performance prediction techniques for proactive system adaptation at run-time.

In the following, we give a conceptual overview of our model-based approach to autonomic performance-aware resource management. In Section 4.1, we introduce the generic MAPE-K adaptation control loop and explain how we refined this concept to integrate and leverage DML to support proactive system adaptation at run-time. In Section 4.2, we present the high-level concepts of DML, our holistic modeling approach with which we aim at describing both static and dynamic aspects of the system and its adaptation process. More specifically, we discuss requirements for the modeling language. The specific design and implementation of DML as well as the technical realization of the adaptation control loop is given in Chapters 5 and 6.

4.1. Model-Based Adaptation Control Loop

According to Brun et al. (2009), essential elements for building self-adaptive systems are feedback loops, as they provide a generic mechanism for self-adaptation. In the software engineering community, such feedback loops conceptually consist of four distinct phases: Collect, Analyze, Decide, and Act (Cheng et al., 2009). This generic view of feedback loops is based on the concepts of the autonomic control loop model established by the autonomic computing community (e.g., cf. IBM Corporation, 2003; Kephart and Chess, 2003). This concept also distinguishes four main phases, Monitor, Analyze, Plan, and Execute, which are similar to the phases of the software engineering feedback loop. Additionally, this loop contains a Knowledge (Base), which is shared by the other parts.

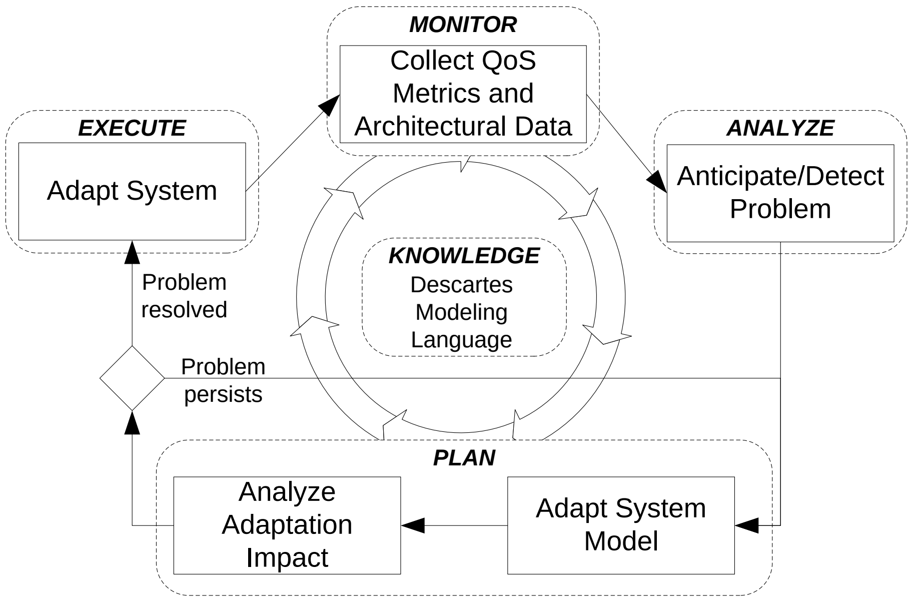

MONITOR
Collect QoS Metrics and Architectural Data
EXECUTE
Adapt System
ANALYZE
Anticipate/Detect Problem
KNOWLEDGE
Descartes Modeling Language
Problem resolved
Problem persists
PLAN
Analyze Adaptation Impact
Adapt System Model

Figure 4.1.: Model-based adaptation control loop.

In the following, we present a refined, model-based version of this generic adaptation control loop concept. Figure 4.1 depicts the four phases of the control loop with the Descartes Modeling Language (DML) serving as a basis for expressing the available Knowledge. The following sections briefly describe the purpose of the control loop phases and how they are linked to DML. In particular, we explain how we integrate DML as a basis for the knowledge base of the adaptation control loop and how we leverage its novel modeling concepts and online performance prediction capabilities (cf. Section 2.2.4) for adaptation decisions at run-time. We also present how we extend the adaptation control loop with an additional feedback loop to support proactive system adaptation.

4.1.1. Monitor

The basis for managing and adapting a system is to know the current state of the system. Thus, the adaptation control loop starts by collecting monitoring data from the managed system. For the performance and resource management purposes of our approach, we are mainly interested in three groups of monitoring data. First, the monitoring data should provide information about the current configuration of the system, i.e., structural information about the hardware infrastructure, execution environment, deployment of virtual machines and services, their used resources, etc. Second, the monitoring data should contain different Quality of Service (QoS) property metrics. In our approach, this includes mainly performance metrics like service response time, throughput or resource utilization, that can be obtained using monitoring frameworks. In addition to performance metrics, it is important to collect further data from the system environment. For example, in our approach we require workload data in the ANALYZE phase to be able to forecast changes in the workload intensity. Third, it is also important to periodically collect information relevant to the goals of the adaptation, e.g., changed customer constraints like Service-Level Agreements (SLAs). Such information is important to ensure that the adapted system is aware of its operational goals.

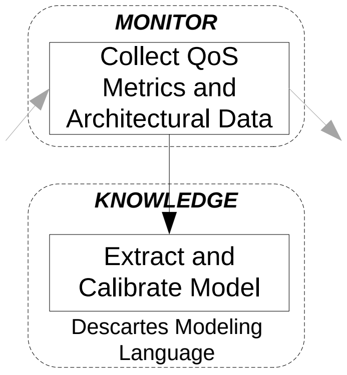

MONITOR
Collect QoS Metrics and Architectural Data
KNOWLEDGE
Extract and Calibrate Model
Descartes Modeling Language

Figure 4.2.: Relation of the monitor phase to the knowledge base.

In general, the monitoring data is persisted in the KNOWLEDGE BASE for reuse in later phases of the adaptation control loop (cf. Figure 4.2). For example, the work by Brosig (2014) uses this monitoring data to create an architecture-level performance model (cf. Section 2.2.1) of the system. This model reflects the software architecture as well as the performance-relevant properties and behavior of the monitored system. Furthermore, Brosig (2014) presents methods to maintain this model instance such that it is up-to-date and reflects the current state of the monitored system accurately. Conceptually, this model can be compared to the concept of a *runtime model*, a model that is causally connected to the system and should provide up-to-date and exact information about the system to drive sub-sequent adaptation decisions (Blair et al., 2009). The methods for the automated extraction and maintenance of such models are presented in Brosig (2014); Brosig et al. (2013, 2009).

4.1.2. Analyze

The general purpose of this phase is to analyze the monitoring data to detect and anticipate violations of the system's operational goals (e.g., SLA violations, inefficient resource usage). If a problem is detected, the system state is analyzed to identify its causes (e.g., a resource bottleneck) such that suitable adaptation strategies can be triggered in the subsequent PLAN phase. This scenario describes an example of *reactive* system adaptation. To enable *proactive* system adaptation, the approach must be able to anticipate performance problems before they have actually occurred. To do this, we need techniques to predict future changes in the system environment. Then, we can apply the predicted changes to our model and analyze the model to detect if the changes will lead to a performance problem.

In this thesis, the performance-relevant changes in the system environment that we focus on are changes of the system's workload. We leverage workload classification and forecasting techniques to predict the workload's future intensity (cf. Chapter 7). We can then use the predicted workload changes as input to online performance prediction techniques (Brosig, 2014) to analyze the impact of these changes on metrics like response time or resource utilization. This allows to detect emerging system bottlenecks or inefficient resource usage and to proactively trigger the search for a solution to the anticipated problem.

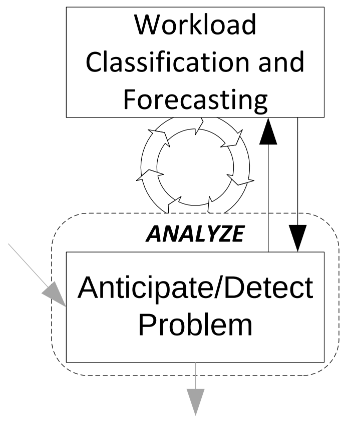

Workload Classification and Forecasting
ANALYZE
Anticipate/Detect Problem

Figure 4.3.: Proactive problem anticipation/detection based on workload classification and forecasting.

The refinement of the general adaptation control loop to support workload forecasting for proactive system adaptation is depicted in Figure 4.1. It consists of an additional feedback loop that runs independent of the overall control loop (cf. Figure 4.3). In this separate loop, we estimate the workload's future intensity.

In summary, the purpose of the Analyze phase is to detect possible performance problems and identify their causes. In the reactive case, the problem analysis can be based on the obtained monitoring data, directly. However, one vital benefit of our model-based approach is that we can leverage the model for proactive system adaptation. In the proactive case, we employ the model and forecasts of changes in the system workload to predict the impact of these changes. These predictions are then used for problem analysis.

4.1.3. Plan

In the Plan phase, we search for a feasible solution to problems identified in the Analyze phase. The term solution in this context refers to a system state in which the anticipated or detected problem has been resolved and that fulfills the system's operational goals. Note that the solution must not necessarily be optimal with respect to the given operational goals. Nevertheless, depending on the specific adaptation options supported by the system and the possible solutions to a detected problem, the search process can become complex and comprise several iterations.

We realize the Plan phase as two main steps (depicted in Figure 4.1) executed iteratively to find a suitable solution to a detected problem. In the first step (*Adapt System Model*), we automatically generate a new system configuration on the model level by applying the adaptation strategy selected as part of the Analyze phase. After applying an adaptation at the model level, we analyze the adapted model (*Analyze Adaptation Impact*) using the online performance prediction techniques of Brosig (2014) to obtain performance metrics for the system state corresponding to the adapted model.

The prediction results are compared with the previous metric values to evaluate the impact of the adaptation, e.g., by comparing response time or resource efficiency metrics. If the applied adaptation is successful, i.e., the detected problem is solved, we can derive a concrete system adaptation plan that is executed on the system in the following Execute phase. If the problem is not solved, the Plan phase continues executing, iteratively generating a different system state while taking into account the determined impact of previously executed adaptations. Note that we pursue an iterative process that aims at improving the managed system to maintain its operational goals and if time is short, this process can be stopped any time.

Conceptually, this phase is independent of specific types of adaptation processes. For each problem detected in the ANALYZE phase, one can imagine different adaptation processes to solve the problem, using simple event-condition-action rules, well-established meta-heuristics (like Tabu Search or Simulated Annealing), or optimization algorithms. To specify such adaptation processes at the model level, DML provides dedicated modeling abstractions. Further details on how to model adaptation processes using DML and about the technical realization will be given in Chapter 6.

4.1.4. Execute

In this phase, we apply the actual adaptation on the real system. To this end, we replay the adaptation actions that have been successfully applied on the model level on the real system. In this phase, the benefit that we have executed all our adaptations on the model level pays off as we can omit all adaptation steps that turned out to lead to an invalid or inadequate system state. To adapt the system and bring it into the desired state, we execute the actions of the concrete adaptation plan derived in the PLAN phase by using the reconfiguration interfaces provided by the real system (e.g., virtualization platforms or middleware).

4.1.5. Knowledge

A central element of the adaptation control loop depicted in Figure 4.1 is the KNOWLEDGE (BASE). Conceptually, the knowledge base can contain any form of information and data required for system adaptation, such as QoS metrics, topology information, or configuration settings (IBM Corporation, 2003). However, an important research question is: What is a suitable approach to represent this knowledge that allows to reason about the system properties but that abstracts from the complexity of the managed system?

The approach presented in this thesis uses the Descartes Modeling Language (DML) to express and capture this knowledge. Briefly, DML provides modeling abstractions to describe the system's performance behavior at the architecture-level, explicitly considering all relevant influences of the system's operational environment and application architecture. Furthermore, DML provides modeling abstractions to capture the system's degrees of freedom and specify adaptation processes at the model level. In the following Section 4.2, we introduce the different modeling abstractions provided by DML and describe the role of DML in our autonomic performance-aware resource management approach.

4.2. Overview of the Descartes Modeling Language

The foundation for our model-based adaptation approach is the Descartes Modeling Language (DML), an architecture-level modeling language developed to support the previously described model-based adaptation control loop. Basically, DML is designed to model Quality of Service (QoS) and resource management related aspects of modern dynamic IT systems, infrastructures and services. In the following, we will present an overview of the structure of DML and explain its major design decisions. Note that some parts of DML are part of the contributions of Brosig (2014) and will therefore be denoted accordingly in the following.

The fundamental goal of DML is to provide a holistic model-based approach that can be used to describe the performance behavior and properties of the system as well as to model the system's dynamic aspects like its configuration space and adaptation processes. The intention is that, using the online performance prediction techniques provided by Brosig (2014), DML can support the ANALYZE phase of the adaptation control loop in problem anticipation and the PLAN phase in autonomic decision-making. Furthermore, by providing means to specify adaptation processes at the model level, DML can be used to find suitable system configurations without having to adapt the actual system. However, these requirements lead to two different concerns that must be addressed by DML. First, DML has to reflect the performance behavior of the managed system. Second, it must be suitable to describe the adaptation process of the system. Thus, the important question is how to separate these concerns (cf. France and Rumpe, 2007, p. 4).

To address this challenge, DML explicitly distinguishes different model types that describe the system and its adaptation processes from a *technical* and a *logical* viewpoint. Together, these different model types form a DML instance (cf. Figure 4.4). The idea of using separate models is to separate knowledge about the system architecture and its performance behavior (technical aspects) from knowledge about the system's adaptation processes (logical aspects).

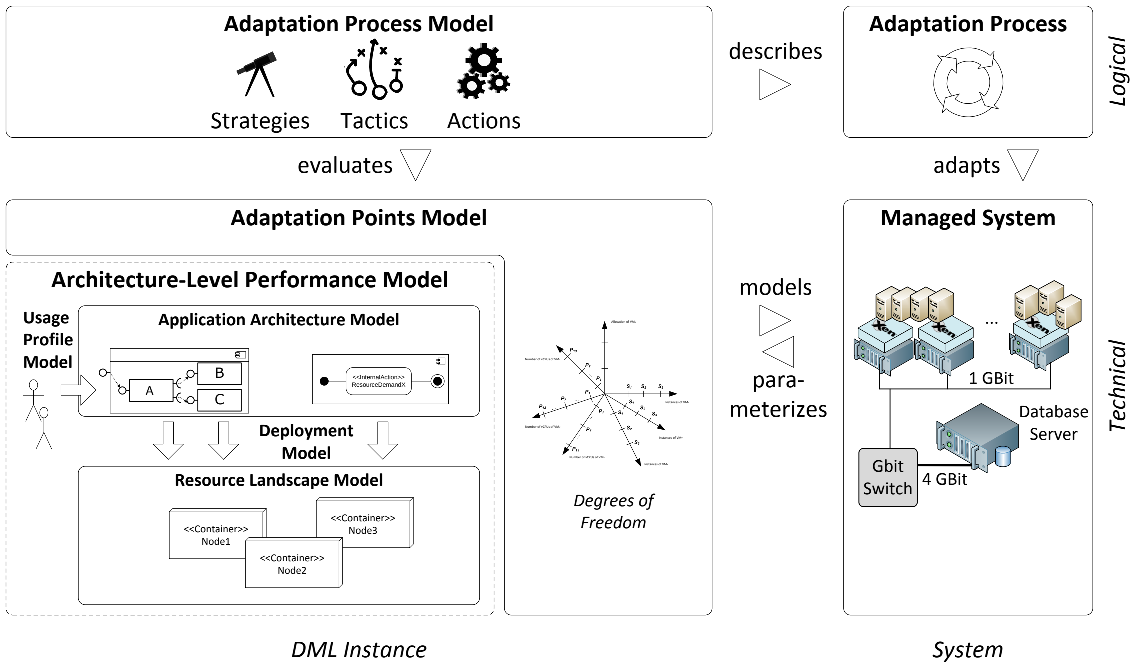


Figure 4.4.: Relation of the different models of a DML instance and the system, separated into technical and logical aspects.

Figure 4.4 depicts an overview of the relation of the different models that are part of a DML instance, the managed system, and the system's adaptation process, from the technical and logical viewpoint. In the bottom right corner of Figure 4.4, we see the system that is managed by a given, usually system-specific, adaptation process, depicted in the top right corner of Figure 4.4. In the bottom left corner, we see the models that reflect the technical aspects of the system relevant for model-based performance and resource management. These aspects are the hardware resources and their distribution (resource landscape model), the software components and their performance-relevant behavior (application architecture model), the deployment of the software components on the hardware (deployment model), the usage behavior and workload of the users of the system (usage profile model), and the degrees of freedom of the system that can be employed for run-time system adaptation (adaptation points model). The resource landscape model, the deployment model, and the adaptation points model are part of the contributions of this thesis. The application architecture model and the usage profile model are part of the work of Brosig (2014).

On top of these models (top left corner of Figure 4.4), we see the adaptation process model that specifies an adaptation process describing how to adapt the managed system. The adaptation process leverages online performance prediction techniques to reason about possible adaptation strategies, tactics, and actions. The adaptation process model is part of the contribution of this thesis.

In the following, we introduce the different model types in more detail and list the major requirements that influenced this design of DML. We look at the different models from a technical and logical viewpoint to clearly separate these aspects.

4.2.1. Technical Viewpoint

To describe and model the managed system from the technical viewpoint, we take the perspective of a system architect. System architects design the system, i.e., they have knowledge about the technical details of the system. This includes the operational environment and the architecture of the managed system as well as functional and non-functional properties of the system. Furthermore, to provide flexibility to adapt the system to changes in its environment, the system architect also knows which entities of the system are intended to be changed dynamically at run-time.

From the technical viewpoint, we first need a model with a causal connection to the managed system (Blair et al., 2009). This type of model provides up-to-date and exact information about the system and is also called *reflection model* (cf. Vogel et al., 2011). As this thesis is focused on performance and resource management aspects, the reflection model should be a model that reflects the performance properties of the system. The reflection model can then be leveraged to analyze the impact of changes in the system environment on the system performance. Furthermore, the reflection model should abstract the system at the architecture level such that we can leverage structural information for adaptation decisions and for describing adaptation processes at the model level. In the following, we introduce four different models targeted at describing the performance behavior of the system, the resource landscape, application architecture, deployment, and usage profile model (cf. Figure 4.4). Together, these models constitute an *architecture-level performance model*, a model that describes the performance behavior of the system at the architecture-level (cf. Section 2.2.1 for more details). This architecture-level performance model is used by Brosig (2014) for online performance prediction (cf. Section 2.2.4) and thus serves as the reflection model in our model-based system adaptation control loop.

Resource Landscape Model

The purpose of this model is to describe the structure and the properties of both physical and logical resources of modern distributed IT service infrastructures. Therefore, the resource landscape model provides modeling abstractions to specify the available physical resources (CPU, network, HDD, memory) as well as their distribution within data centers (servers, racks, and so on). To specify the logical resources, the resource landscape model also supports modeling different layers of resources and specifying the performance influences of the configuration of these layers. In this context, resource layers denote the software stack on which software is executed, including virtualization, operating system, middleware, and runtime environments (e.g., JVM). In addition, as we also consider systems distributed over multiple data centers, the model also captures the distribution of resources across data centers. Modeling the structure and properties of data center resources at this level of detail is important for accurate performance predictions and to derive causal relationships of the performance impact during system adaptation. This model is one of the main contributions of this thesis and explained in detail in Chapter 5.

Application Architecture Model

This model is focused on the application architecture of the managed system. For performance analysis, this model must capture performance-relevant information about the software services that are executed on the system as well as external services used by the system. In general, this model is focused on describing the performance behavior of the software services after the principles of component-based software systems (Becker et al., 2009). A software component is defined as a unit of composition with explicitly defined provided and required interfaces (Szyperski et al., 2002). The performance behavior of each software component can be described independently and at different levels of granularity. The supported levels of granularity range from black-box abstractions (a probabilistic representation of the service response time behavior), over coarse-grained representations (capturing the service behavior as observed from the outside at the component boundaries, e.g., frequencies of external service calls and amount of consumed resources), to fine-grained representations (capturing the service's internal control flow and internal resource demands). The advantage of the support for multiple abstraction levels is that the model is usable in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to detailed system simulation. Moreover, one can select an appropriate abstraction level to match the granularity of information that can be obtained through monitoring tools at run-time, e.g., considering to what extent component-internal information can be obtained by the available tools. The application architecture meta-model is developed as part of the thesis of Brosig (2014). An overview of the concepts relevant to this thesis is given in Section 5.3.1.

Deployment Model

To analyze the performance of the modeled system, it is necessary to connect the modeled software components with the system resources described using the resource landscape model. The deployment model provides this information by mapping the software components modeled in the application architecture model to physical or logical resources described in the resource landscape model. With this mapping, resource demands of the modeled software components can be traced through the layers of the resource landscape model down to the physical resources. Thereby, it is possible to analyze mutual performance influences when sharing resources. A detailed description of this model is given in Section 5.3.3.

Usage Profile Model

Finally, an important aspect that influences the performance of a system is the way the system is used. For instance, if the amount of user requests that have to be processed by the system increases, more resources would normally be required to process the increased amount of work. The usage profile model can be used to describe the types of requests that are processed by the system and the frequency with which new requests arrive. In fact, the usage profile is a frequently changing property of the system environment to which we want to adapt the system proactively. A description of this model is given in Section 5.3.2.

Together, these four models form an architecture-level performance model that conveys detailed information about the structural and performance behavior of the system. During the ANALYZE and PLAN phases, we can leverage these aspects to for proactive system adaptation and to support reasoning about adequate adaptation decisions.

However, to describe adaptation processes at the model level and to decide how to adapt the system at run-time, we need additional modeling abstractions to specify the degrees of freedom of the system, i.e., the parts of the system that can be changed at run-time to adapt the system to changes in its environment. Vogel et al. (2011) refer to such mode types as *change model*, as part of a larger *adaptation model*. DML provides such an additional change model (called adaptation points model, cf. Figure 4.4). This adaptation points model annotates the architecture-level performance model to describe the degrees of freedom of the system architecture (cf. Figure 4.4).

Adaptation Points Model

This model provides modeling abstractions to describe the elements of the resource landscape and the application architecture that can be leveraged for adaptation (i.e., reconfiguration) at run-time. Other model elements that may change at run-time but cannot be directly controlled (e.g., the usage profile), are not in the focus of this model. Adaptation points on the model level correspond to operations that can be executed on the system at run-time to adapt the system (e.g., adding virtual CPUs (vCPUs) to VMs, migrating VMs or software components, or load-balancing requests). Thus, the adaptation points model defines the configuration space of the managed system. The model provides constructs to specify the degrees of freedom along which the system's state can vary as well as to define boundaries for the valid system states. This model is part of the contributions of this thesis and it is introduced in detail in Section 6.1

The purpose of having a model that explicitly describes the adaptation points of the system is to support separating knowledge of technical system aspects from logical adaptation aspects (cf. Figure 4.4). By defining explicit adaptation points, we can reuse them to specify adaptation processes on the model level based on the given adaptation points. Thereby, the modeled adaptation process is bounded by the specified degrees of freedom of the system but it is independent of system-specific details. In the next section, we introduce how we describe the logical aspects of system adaptation at the model level.

4.2.2. Logical Viewpoint

In general, self-adaptive systems follow certain automatic or semi-automatic processes to adapt to changes in their environment such that their operational goals are continuously satisfied. These processes are usually implemented by a software entity, often referred to as agent (cf. Franklin and Graesser, 1997). Such agents are responsible for controlling the system adaptation processes. Usually, they encapsulate adaptation logic that can be expressed in the form of simple rules or using complex heuristics or optimization algorithms. The implementations of adaptation logic is usually highly system specific and thus difficult to reuse in different contexts. To address this problem, DML provides a domain-specific modeling language that abstracts from these technical details. Thereby, adaptation processes can be specified at the model level, defining when, where, and how the managed system should be adapted.

Adaptation Process Model

This model can be used to describe processes that keep the system in a state such that its operational goals are continuously fulfilled, i.e., it describes the way the system adapts to changes in its environment. It is based on the previously introduced architecture-level performance model and adaptation points model which are used to describe adaptation processes at the model level. With this model, we aim at abstracting from technical details such that we can describe adaptation processes from a logical perspective, independent of system-specific details. It is designed to provide sufficient flexibility to model a large variety of adaptation processes from event-condition-action rules to complex algorithms and heuristics. Essentially, it distinguishes high-level goal-oriented objectives, adaptation strategies and tactics, from low-level system-specific adaptation actions. The modeling language also provides concepts to describe the operational goals of the managed system such that the adaptation process can be driven towards these goals. This model is a core contribution of this thesis and its concepts are explained in detail in Section 6.2.

4.3. Summary

In this chapter, we introduced the two major conceptual building blocks of our model-based adaptation approach. First, we presented a refined concept of the MAPE-K adaptation control loop designed to leverage the novel features of the Descartes Modeling Language (DML) to realize autonomic performance-aware resource management at run-time. Second, we introduced the major concepts of DML that we employ for online performance prediction and run-time system adaptation.

In summary, with our modeling approach, we provide a clear separation of the technical and logical concerns when engineering self-adaptive systems. To model the technical and performance-related aspects of a self-adaptive system, we use an architecture-level performance model, annotated by an adaptation points model to describe the system's degrees of freedom. Our model-based approach is completed by a modeling language that can be used to describe logical aspects of adaptation processes at the model level. Thus, with DML, we provide a holistic model-based approach for autonomic performance-aware resource management.

In the remainder of this thesis, we elaborate the conceptual details of DML. In Chapter 5, we present the modeling abstractions to describe the resource landscape, application architecture, deployment, and usage profile of the adapted system. Then, Chapter 6 presents the modeling abstractions to describe the adaptation points of the managed system as well as a modeling language to specify adaptation processes. In Section 6.3, we present the realization of our model-based adaptation control loop using DML and its online performance prediction capabilities (cf. Brosig, 2014). Finally, in Chapter 7, we show how to use workload classification and forecasting techniques to enable proactive system adaptation.

5. Modeling System Resource Landscapes and Their Performance Influences

Koziolek (2010) distinguishes four major types of aspects that are important when modeling the performance of modern IT systems: the usage profile of the system, the implementation of the software components, the performance of external services used by the system, and the execution environment. Current architecture-level performance models as surveyed by Balsamo et al. (2004) and Koziolek (2010) typically describe the system's infrastructure and platform as a flat hierarchy of resources. Such models usually abstract the detailed structure of resources and do not explicitly capture the impact of possibly multiple resource layers like virtualization or middleware. However, for autonomic system adaptation and resource management at run-time, such information is crucial to analyze the possible impact of adaptation operations (like migrating VMs or services) on the system performance.

In this chapter, we present a modeling language to describe the resource landscape of modern distributed IT systems. In Section 5.1, we introduce novel concepts for modeling the configuration of physical resources as well as the performance-influencing properties of resource layers. In Section 5.2, we present a method for the automatic quantification of such performance-influencing properties using virtualization as proof-of-concept. In Section 5.4, we present two case studies. The first case study demonstrates how resource landscape models can be leveraged to improve run-time resource management. The second case study evaluates the performance overhead of two representative virtualization platforms, Citrix XenServer 5.5 and VMware ESX 4.0.

Other important aspects that influence the performance of modern IT systems are the software system's design and implementation, the deployment of the software components on the resource landscape, and the usage profile. Meta-models to describe these aspects are also part of DML (cf. Chapter 4). However, as they are not in the focus this thesis, we only give a brief overview of their main concepts (Section 5.3) that are relevant for our automated system adaptation and resource management approach. More details on how this meta-models can be used for online performance prediction are part of the work of Brosig (2014).

5.1. Resource Landscape Meta-Model

The motivation for our resource landscape meta-model is to provide novel modeling abstractions that can be used to describe the complex nature of modern distributed IT infrastructures. Briefly, these novel constructs support modeling the distribution of resources within and across the boundaries of data centers, the nested layers of resources, and the performance influences of the different resource layers. They are explained in more detail in the following sections.

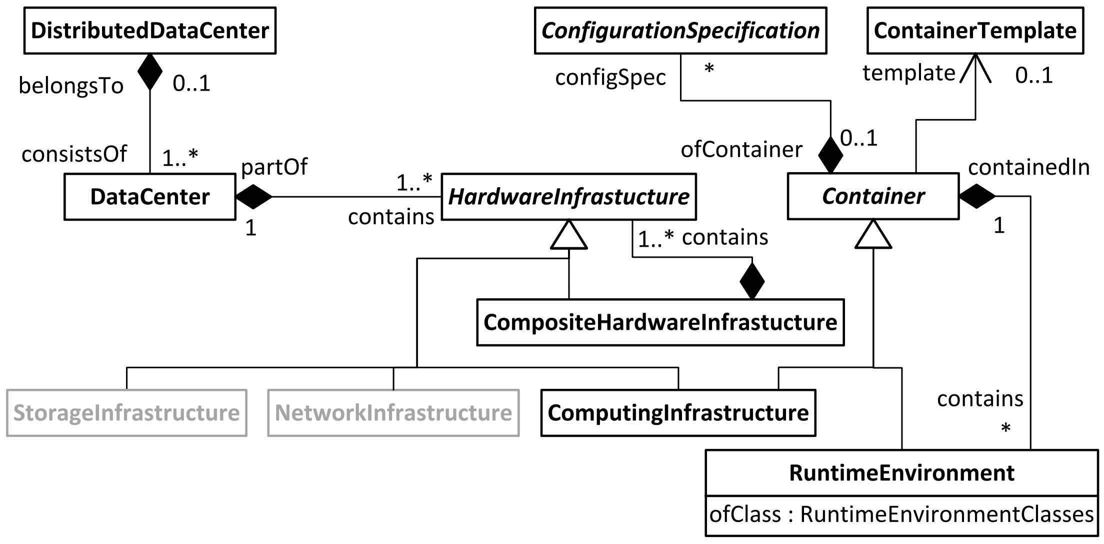


Figure 5.1.: The resource landscape meta-model.

Figure 5.1 depicts an overview of the structure of the resource landscape meta-model as a UML class diagram. The root entity comprising all other model elements is the DistributedDataCenter, which consists of one or more DataCenters. DataCenters contain HardwareInfrastructures which are either one of the three hardware infrastructure types ComputingInfrastructure, NetworkInfrastructure, and StorageInfrastructure, or CompositeHardwareInfrastructure. A CompositeHardwareInfrastructure is a structuring element to group further HardwareInfrastructures. For example, it can be used to combine servers to a cluster or to group them in a server rack. Current architecture-level performance models usually abstract from these details and do not provide constructs to model the resource hierarchy and containment relationships explicitly. However, for resource management at run-time, the description of the resource landscape and hierarchy of resources is crucial to improve reasoning about suitable adaptation operations, e.g., to decide if a VM can be migrated and where it should be migrated to. Here, the novel aspect of our meta-model is that it allows to model the distribution of resources within and across data centers as well as their individual configuration.

When designing the resource landscape meta-model, we aimed at a generic approach to cover all types of infrastructure with the focus on ComputingInfrastructure. More details on modeling storage and network infrastructures (i.e., the StorageInfrastructure and NetworkInfrastructure entities in the meta-model) can be found in the work of Noorshams et al. (2013a,b,c) and Rygielski et al. (2013a,b), respectively.

5.1.1. Containers and Containment Relationships

A common reappearing pattern in modern distributed IT service infrastructures is the nested containment of system entities, e.g., data centers contain servers, servers typically contain a set of virtual machines (VMs) hosted on a virtualization platform, servers and VMs run an operating system, which may contain a middleware layer, and so on. This leads to a tree of nested system entities that may change during runtime because of virtual machine migration, hardware or software failures, etc. The central element of our resource landscape meta-model to model these nested layers of resources is the abstract entity

Container, depicted in Figure 5.1. We distinguish between two major concrete container entities: the ComputingInfrastructure and the RuntimeEnvironment. The ComputingInfrastructure forms the root element in our hierarchy of containers and corresponds to a physical machine within a data center. This entity cannot be contained in another container, but it may have nested containers (RuntimeEnvironments). The RuntimeEnvironment is the second type of container. It can contain further RuntimeEnvironments. Thereby, we realize the modeling of nested containers and can create a hierarchy of resources.

Furthermore, each RuntimeEnvironment has the property ofClass to specify the class of the RuntimeEnvironment. A Container has a property configSpec to specify its ConfigurationSpecification and a property template referring to a ContainerTemplate. These concepts will be explained in the following sections.

5.1.2. Classes of Runtime Environments

We distinguish several general classes of runtime environments which are listed in Figure 5.2: HYPERVISOR for the different hypervisors of virtualization platforms, OS for operating systems, OS_VM for virtual machines emulating standard hardware, PROCESS_VM for process virtual machines like the Java VM or the Common Language Runtime (CLR), MIDDLEWARE for middleware environments, and OTHER for any other type. This list can be extended if new classes are required. The purpose of distinguishing different classes of runtime environments is to constrain the possible combinations of runtime environments within the hierarchy. By setting the ofClass property of the RuntimeEnvironment to one of these values, we can specify OCL constraints to enforce consistency within the modeled layers. To prohibit the instantiation of different RuntimeEnvironment classes within the same container, we specify the following OCL constraint:

```
context RuntimeEnvironment
inv runtimeEnvironmentLevelCompliance:
    self.containedIn.contains
        ->forAll(r : RuntimeEnvironment | r.ofClass = self.ofClass);
```

Listing 5.1: OCL invariant checking RuntimeEnvironment compliance.

As a result, a RuntimeEnvironment can only contain containers that are of the same class, e.g., a hypervisor can only contain virtual machines and not further hypervisors.

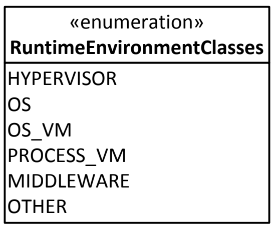


Figure 5.2.: Different runtime environment classes.

Another solution would have been to model the different types of runtime environments as explicit entities. However, we wanted to design a model that is easy to extend. Modeling all classes of runtime environments as explicit model entities would have required to also explicitly model their relations (e.g., OS_VM can only be contained in HYPERVISOR) which makes the meta-model much more complex and difficult to maintain. By using the ofClass attribute and the RuntimeEnvironmentClasses, new classes can be introduced by extending the enumeration, which has less impact on the meta-model structure. This makes it easier to reuse and extend the model instances. Also, we can assume that model instances can be built automatically or with tool support and that the tool support will automatically enforce such constraints.

5.1.3. Resource Configuration Specification

Each Container has its own specific resource configurations that describe the container's influence on the system performance. In our meta-model, we distinguish between three different types of configuration specifications: ActiveResourceSpecification, PassiveResourceSpecification, and CustomConfigurationSpecification, depicted in Figure 5.3.

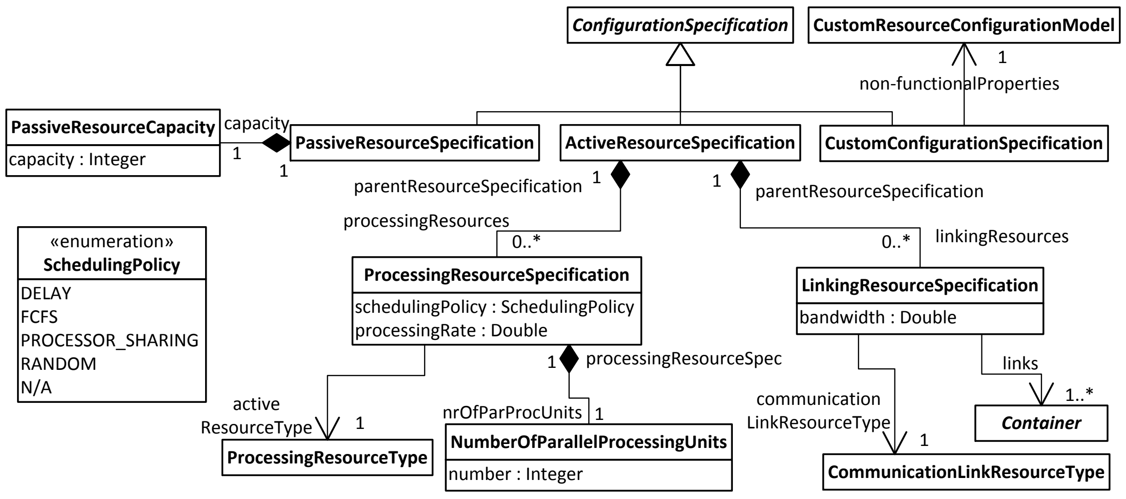


Figure 5.3.: Types of resource configurations.

The ActiveResourceSpecification can be used to specify the active resources of a Container. Active resources can actively execute a task. Examples for active resources are CPUs, hard disks, and network connections. We further distinguish between ProcessingResourceSpecifications and LinkingResourceSpecifications. The ProcessingResourceSpecification can be used to specify what ProcessingResourceTypes the modeled entity offers. The currently supported ProcessingResourceTypes are CPU and HDD. The ProcessingResourceSpecification is further defined by its properties schedulingPolicy and processingRate. These parameters influence the time the active resource needs to process a task. For example, a CPU can be specified with PROCESSOR_SHARING as schedulingPolicy and a processingRate of 2.66 GHz. If a ProcessingResourceSpecification has more than one processing units (e.g. a CPU has four cores), the attribute number of the entity NumberOfParallelProcessingUnits would be set accordingly, whereas two CPUs would be modeled as two separate ProcessingResourceSpecifications. The LinkingResourceSpecification can be used to describe communication links between containers on a high level of abstraction. For a more detailed modeling of the performance-relevant aspects of the network including network interface cards, routers, switches, and so on, we refer to the work of Rygielski et al. (2013b). In this thesis, a LinkingResourceSpecification abstracts from such details and describes only the communication links between the source container and the target containers. The source container is the container the LinkingResourceSpecification belongs to. The links to the target containers are characterized by a shared bandwidth and a CommunicationLinkResourceType. The currently supported CommunicationLinkResourceType is LAN.

The PassiveResourceSpecification can be used to specify properties of passive resources such as semaphores, threads, monitors, etc. Passive resources are not able to process requests. They usually have a limited capacity which can only be acquired and released. Examples for passive resources are the main memory size, the number of database connections, the heap size of a JVM, or resources in software like thread pools, etc. Passive resources refer to a PassiveResourceCapacity, the parameter to specify the size of the passive resource, e.g., the number of threads or memory size.

If the concepts of ActiveResourceSpecification or PassiveResourceSpecification are not sufficient to model the resource configurations of more complex resources (e.g., a hypervisor), one can use the CustomConfigurationSpecification. This model entity refers to the abstract class CustomResourceConfigurationModel which severs as a placeholder for any other custom model that can be used to describe performance-relevant resource configuration properties. Instances of CustomResourceConfigurationModels can then be employed during online performance analysis to consider the performance-relevant properties for this resource. In Section 5.2, we present an example CustomResourceConfigurationModel that describes the performance-relevant resource configuration for hypervisors with feature models.

5.1.4. Container Types

With the modeling abstractions presented so far, it is necessary to model each container and its resource configuration specification explicitly (cf. left-hand side of Figure 5.4). This can be very time-consuming when creating large model instances, especially when modeling clusters of several hundred identical machines. Hence, a concept to specify the multiplicity of model entities can improve model usability and comprehensiveness. However, while with multiplicities one can specify the number of instances of a model entity, the different instances are indistinguishable and would all have the same attribute values. This is a problem since in data centers, there might exist multiple instances of the same container type but with different resource configurations, i.e., they must be distinguishable. For example, Virtual Machines (VMs) of the same customer can have a similar default resource configuration specification, i.e., they are of the same type (cf. Figure 5.4). However, at run-time, we must be able to distinguish the concrete VM instances because their resource configuration specification might change.

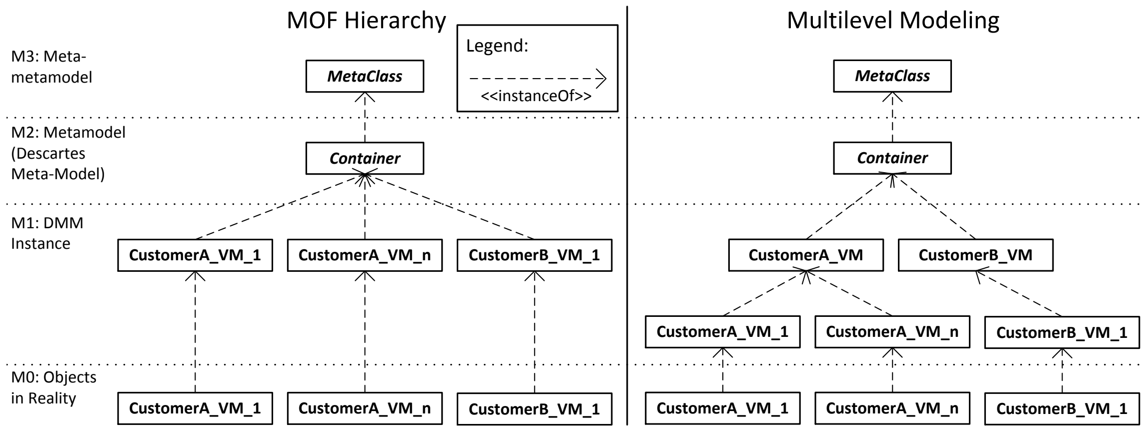


Figure 5.4.: Container instances in the MOF modeling hierarchy (left) and with multilevel modeling (right).

A conceptually elegant solution to address this problem is the multilevel language engineering approach by Atkinson et al. (2009). This approach introduces additional levels in the Meta Object Facility (MOF) specified by the Object Management Group (OMG) (2011b). This way, an instance of a Container can serve as a type for another instance, i.e., it can be instantiated again (cf. right part of Figure 5.4). With Melanie (Atkinson and Gerbig, 2012), there exists a tool based on the Eclipse Modeling Framework (EMF) for multilevel modeling. However, for us this approach was not practical since there is still a fundamental difference between the three-level architecture of EMF, which we use to realize our approach, and the multilevel modeling concepts.

Another solution would have been to develop a second meta-model for modeling container types. This meta-model would act as a "decorator model", i.e., it would extend a resource landscape model instance. The drawback of this solution is that this would introduce a further level of meta-modeling, i.e., an additional meta-model to create instances of container types and thus, container providers (e.g., virtualization platform vendors) must be familiar with meta-modeling.

For these reasons, we decided to implement a hybrid approach and use ContainerTemplates to specify the resource configuration of similar container types. All container instances that are of the same type refer to their container template. These templates are collected in separate model ContainerRepository (see Figure 5.5). Like a Container, the ContainerTemplate also refers to a ConfigurationSpecification to specify the resource configuration for the ContainerTemplate. A Container instance in the resource landscape model can then refer to a ContainerTemplate as its resource configuration specification (see Figure 5.1). We refer to this as a hybrid approach as it supports both ways of modeling containers, either with templates for a group of container type instances or an individual instance for each container.

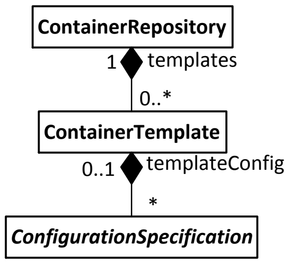


Figure 5.5.: The container templates repository.

The advantage of this modeling approach is that the general resource configuration specifications relevant for all instances of one container type can be stored in the container template. Instance-specific resource configurations deviating from the used container template can still be stored in the individual container instance. This way, only differences to the container template must be modeled and not all individual resource configurations for all different containers. As the container repository is a separate model instance, it can also be reused in other resource landscape models. More formally, let

$R = \{r_1, r_2, \dots, r_n\}$ be the set of resources of a container C.

Furthermore, let $RS = I \cup T$ be the set of resource configuration specifications for the resources of the container C with

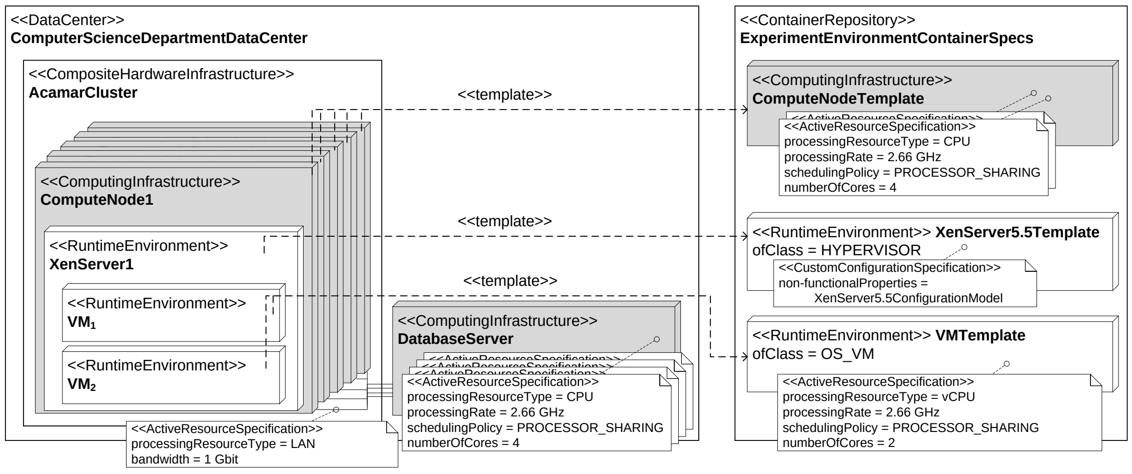
T as the set of *template*-specific resource configuration specifications and
I as the set of *individual* resource configuration specifications.

We assume that for each resource $r \in R$, there exists a resource configuration specification.

Then, during model analysis, the semantic of a container template is the following: If a container has no individual resource configuration specification $i_j \in I$ for resource $r_j \in R$, it inherits the specification from its referenced container template resource configuration specifications T, i.e., $rs_j = t_j$. If a container defines its own individual resource configuration specification $i_j \in I$, the latter overrides the resource configuration specification of the template $t_j \in T$, i.e., $rs_j = i_j$. For example, as a container, assume a VM with two processing resources and one networking resource. The resource configuration specification of the container template of this VM is $T_{VM} = \{t_1, t_2, t_3\}$. Therefore, the VM inherits the initial resource configuration specifications of its template, i.e., $RS_{VM} = \{t_1, t_2, t_3\}$. If we change the resource configuration specification of one resource, e.g., because we add a virtual CPU to the second processing resource, the new set of resource configuration specifications is $RS_{VM} = \{t_1, i_2, t_3\}$

5.1.5. Example Resource Landscape Model Instance

To illustrate our meta-model concepts, we use an example model instance of the resource landscape from our cluster environment which we later use for validation. Figure 5.6 depicts a resource landscape model instance in a UML-like notation, showing the hierarchy of the different resources as well as their configuration templates.



<<DataCenter>>
ComputerScienceDepartmentDataCenter
<<CompositeHardwareInfrastructure>>
AcamarCluster
<<ComputingInfrastructure>>
ComputeNode1
<<RuntimeEnvironment>>
XenServer1
<<RuntimeEnvironment>>
VM1
<<RuntimeEnvironment>>
VM2
<<ActiveResourceSpecification>>
processingResourceType = LAN
bandwidth = 1 Gbit
<<template>>
<<template>>
<<template>>
<<ComputingInfrastructure>>
DatabaseServer
<<ActiveResourceSpecification>>
processingResourceType = CPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 4
<<ContainerRepository>>
ExperimentEnvironmentContainerSpecs
<<ComputingInfrastructure>>
ComputeNodeTemplate
<<ActiveResourceSpecification>>
processingResourceType = CPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 4
<<RuntimeEnvironment>> XenServer5.5Template
ofClass = HYPERVISOR
<<CustomConfigurationSpecification>>
non-functionalProperties =
XenServer5.5ConfigurationModel
<<RuntimeEnvironment>> VMTemplate
ofClass = OS_VM
<<ActiveResourceSpecification>>
processingResourceType = vCPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 2

Figure 5.6.: Example resource landscape model instance.

The root element is the local DataCenter in our computer science department at KIT, which contains a CompositeHardwareInfrastructure (a cluster environment called *AcamarCluster*), and a separate ComputingInfrastructure, the *DatabaseServer*. The cluster in this example consists of five *ComputeNodes*, connected by a 1 Gbit Ethernet LAN. Each compute node runs XenServer 5.5 as a hypervisor. On top of each XenServer, we execute two VMs. The *DatabaseServer* is a separate machine, connected to the cluster with four 1 Gbit Ethernet connections. It has four six-core CPUs with 2.66 GHz and PROCESSOR_SHARING as scheduling policy. To ease the resource configuration specification of the other containers, we use the container template mechanism of the resource landscape meta-model.

The resource configuration specification templates for the different container types are stored in the *ExperimentEnvironmentContainerSpecs* container repository. The *ComputeNodeTemplate* specifies the hardware resource configuration of the cluster compute nodes. A compute node has two ActiveResourceSpecifications modeling its two CPUs. Each has four cores with 2.66 GHz and PROCESSOR_SHARING as scheduling policy. The *XenServer5.5Template* is a template for a RuntimeEnvironment of class HYPERVISOR. It refers to a CustomConfigurationSpecification, which refers to a CustomResourceConfigurationModel for the XenServer 5.5 hypervisor. Further details of this custom model will be presented in the following Section 5.2.1 when we discuss the performance influences of the various resource layers. Finally, the *VMTemplate* specifies the configuration of the VMs hosted by the XenServer. This RuntimeEnvironment is of class OS_VM and has only one ActiveResourceSpecification for its virtual CPU (vCPU). It has two cores with 2.66 GHz and PROCESSOR_SHARING as scheduling policy.

5.2. Performance-Influencing Factors of Resource Layers

Modern IT systems can have a complex stack of resource layers. In this context, we use the term *resource layer* to denote any software layer that abstracts from underlying physical resources and introduces a new layer of virtual resources. Such resource layers (e.g., virtualization, middleware, process virtual machines) enable resource sharing through a flexible mapping of virtual to physical resources. Figure 5.7 depicts an example in which two components run in a middleware environment on top of virtualized hardware. The drawback of having multiple resource layers is that each layer can have its own specific impact on the system performance, ranging from minor additional management overhead, e.g., due to scheduling, to major interferences due to the sharing of the underlying physical resources. These influences and their extent usually depend on the configuration of the different resource layers.

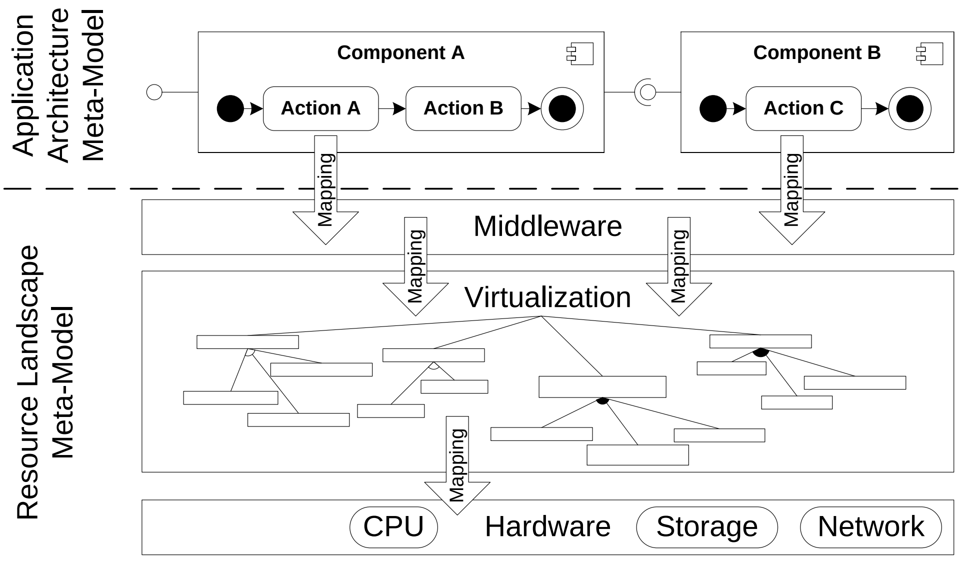


Figure 5.7.: Schematic overview of resource abstraction layers.

To consider the effect of such influences in the performance analysis process, it is necessary to analyze, quantify, and integrate them in the performance model. In the following sections, we introduce a method to classify, quantify, and model performance-influencing factors of resources layers, using virtualization platforms as a proof-of-concept. First, we identify and classify the performance-influencing factors of virtualization platforms (Section 5.2.1). Based on this classification, we conduct automated experiments to quantify the influences of the identified factors. The results of these experiments are presented in detail in Section 5.2.2. From these results, we derive a performance model (cf. Section 5.2.3) that can be employed in the online performance prediction process to improve the prediction accuracy of architecture-level performance models.

5.2.1. Classification of Performance-Influencing Factors

In this section, we classify the performance-influencing factors of state-of-the-art virtualization platforms, listed in Table 5.1. We focus on those factors that have a considerable impact on the system performance, i.e., they should be considered in the performance analysis process. The set of factors we consider as important is based on studies available in the literature (Barham et al., 2003; Apparao et al., 2006; Padala et al., 2007; Quétier et al., 2007; Soltesz et al., 2007; Tickoo et al., 2010; VMware, 2007) and on our experiences with virtualization gained in the last years. The goal of our classification is to provide a compact hierarchical model of the performance-influencing properties and their dependencies.

We structure these factors in a so-called *feature model* (Czarnecki and Eisenecker, 2000). In our context, a *feature* corresponds to a performance-relevant property or a configuration option of the considered virtualization platform. The goal of the feature model is to organize the options that have an influence on the performance of the virtualization platform in a hierarchical structure. Furthermore, the feature model also considers external influencing factors such as workload profile or type of hardware (e.g., hardware with or without virtualization support). The model we propose is depicted in Figure 5.8.

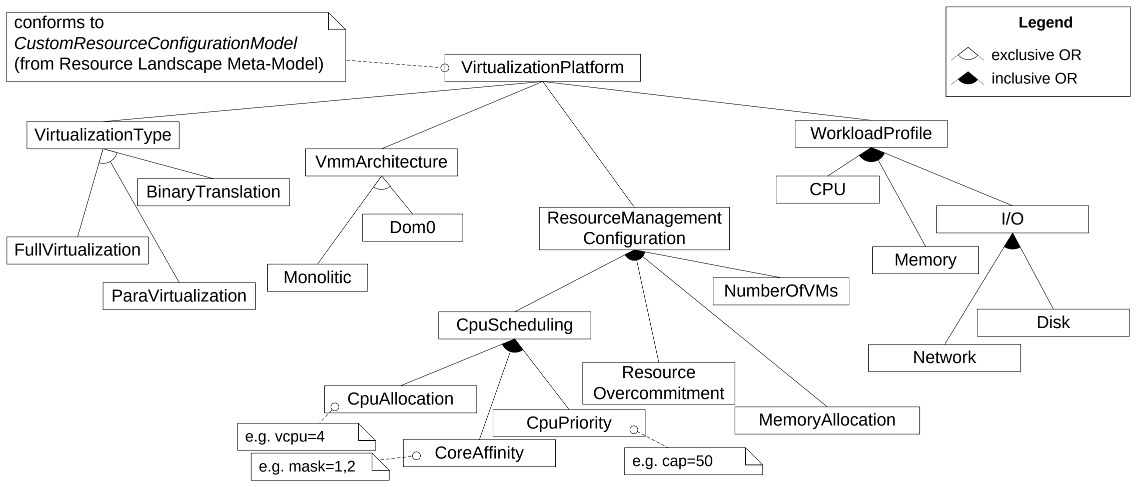


Figure 5.8.: Major performance-influencing factors of virtualization platforms.

The first performance-influencing factor is the VirtualizationType. Different virtualization techniques might cause different performance overhead, e.g., full virtualization performs better than other alternatives because of the hardware support. In our feature model, we distinguish between the three major types of virtualization: FullVirtualization, ParaVirtualization, and BinaryTranslation (cf. Huber et al. (2010b) for an introduction to the different virtualization approaches). Furthermore, another important performance-influencing factor is the hypervisor's architecture (VmmArchitecture). The two options here are monolithic architectures, like in VMware ESX/ESXi, or architectures with a dedicated control domain (Dom0), like in Xen.

The ResourceManagementConfiguration groups several influencing factors. First, the CpuScheduling configuration has a significant influence on the virtualization platform's performance and is influenced by several factors. The first factor CpuAllocation reflects the number of virtual CPUs allocated to a VM. Most of the performance loss of Central Processing Unit (CPU) intensive workloads comes from core and cache inferences (Apparao et al., 2008; Huber et al., 2011b). Hence, the second factor is CoreAffinity, specifying if virtual CPUs of VMs are assigned to dedicated physical cores (core-pinning). The third

Table 5.1.: Common virtualization platforms (PV = para-virtualization, FV = full virtualization).

Name	Supports	Executed On	License	Since
Xen	PV/FV	bare metal	GPL	09-2003
KVM	FV	bare metal	(L)GPL (v2+)	02-2007
VirtualBox	FV	OS	GPL	02-2007
VMware ESX/ESXi	FV/PV	bare metal	commercial	12-2007

factor reflects the capability of assigning different CpuPriorities to VMs. For example, the Xen hypervisor's cap parameter or VMware's limits and fixed reservations parameters are CPU priority configurations. In addition, the level of ResourceOvercommitment influences the performance due to contention effects caused by resource sharing. Finally, the MemoryAllocation and the NumberOfVMs influence the resource management configuration, too. Managing virtual memory requires an additional management layer in the hypervisor. The number of VMs has a direct effect on how the available resources are shared among them.

From an external point of view, an important influencing factor is the WorkloadProfile executed on the virtualization platform. Virtualizing different types of resources causes different performance overheads. For example, CPU virtualization is supported very well whereas I/O and memory virtualization currently suffer from significant performance overheads. In our model, we distinguish CPU, Memory and I/O intensive workloads. In the case of I/O workload, we further distinguish between Disk and Network intensive I/O workloads. Of course, one can also imagine a workload mixture as a combination of the basic workload types.

5.2.2. Automatic Quantification of Performance-Influencing Factors

For some resource layers, a manual quantification of the effect of the various performance-influencing factors is too expensive, especially in scenarios with a large parameter space. Therefore, we developed a method based on automated experimental analysis to quantify the performance influences of the identified factors. In the following, we describe a prototypical implementation of this method for virtualization platforms. We used this implementation to automatically conduct experiments (Section 5.2.2.1) that quantify the performance influences captured in our virtualization platform feature model. We then describe the different types of experiments (Section 5.2.2.2) that are implemented in the process and how to structure them to assess the performance influence of a given factor. In Section 5.2.2.3, we provide an overview of several benchmarks that we use for evaluating the various influence factors. Section 5.2.2.4 gives an overview of the hardware environments that we use for our experimental analysis. Finally, in Section 5.2.2.5, we present experiment results for Citrix's XenServer 5.5 in these hardware environments.

5.2.2.1. Method

The purpose of this method is to reduce the manual overhead for conducting experiments by automating the configuration of the VM and/or benchmark parameters, starting, cloning, and stopping VMs, and the collection and analysis of the benchmark results. As a first step of this method, we install the virtualization platform to be evaluated on the target hardware platform. Next, we create a *MasterVM* (see Figure 5.9) which serves as a template for creating multiple VM clones that will later execute a selected benchmark. To this end, the respective benchmark is installed on the MasterVM together with scripts to control the benchmark execution (e.g., to schedule benchmark runs). The MasterVM is the only VM with an external network connection. All other VMs and the MasterVM are connected via an internal network. The second important part of our method is the *Controller* which runs on a separate machine. It adjusts the configuration (e.g., amount of virtual CPUs) of the MasterVM and the created VM clones as required by the considered type of experiments. The controller also clones, deletes, starts, and stops VMs via the virtualization layer's API. Furthermore, it is responsible for collecting, processing and visualizing the results. In the presented method, the benchmark choice is left open and one can use any available benchmark that can measure and quantify the performance-influencing factor under consideration. In Section 5.2.2.3, we present an overview of several benchmarks that we recommend for the various types of workloads.

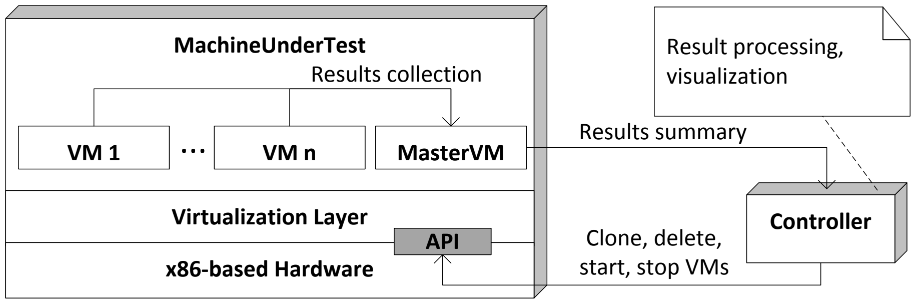


Figure 5.9.: Schematic view of the experimental setup.

Figure 5.10 shows the process of automated execution of experiments from the controller's point of view. At first, the controller starts the MasterVM and then configures the benchmark to be executed and schedules the experiment runs. After that, the controller replicates the MasterVM according to the requirements of the respective set of experiments described in Section 5.2.2.2. Next, the MasterVM must be stopped because it only serves as a template and it is not intended to run during the experiments. After the VM cloning, the controller performs further VM-specific configurations for each created clone as required by the experiment type, e.g., assigning the VMs' virtual CPUs to physical cores. Finally, the controller starts the VMs and the benchmarks are executed at the scheduled starting time. The controller is responsible to detect the end of the benchmark runs and after the experiments are finished, it triggers the MasterVM to collect the results of all VMs. This is done by the MasterVM because it is the only connection between the VM subnet and the controller. If there are further experiments to be executed, the MasterVM is reconfigured and the whole process starts again from the beginning, continuing until all experiments are completed. Finally, the controller processes and stores the results from the experiments. The gray areas of Figure 5.10 depict the parts of the process where configuration is applied depending on the specific set of experiments considered.

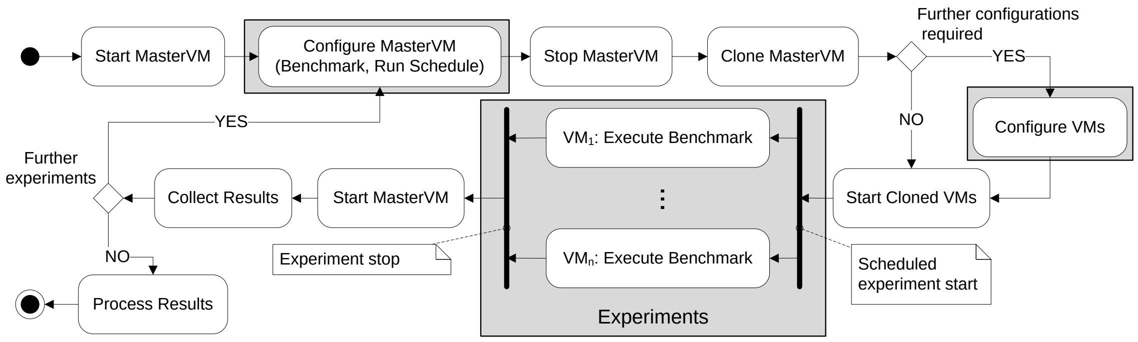


Figure 5.10.: Automated execution of experiments from the controller's point of view.

5.2.2.2. Experiment Types

We now describe different experiment types to quantify the effect of the performance-influencing factors we described in the feature model in Section 5.2.1. The feature model distinguishes between the following categories of influencing factors: (a) virtualization type, (b) VMM architecture, (c) resource management configuration, and (d) workload profile. For category (a) and (b), different virtualization platforms must be installed on the experiment hardware to quantify the performance overhead of the different platforms, or a plain operating system to compare a virtualized platform with a native system. Experiment types for these categories are not supported by our automation. However, after installing a native operating system or a virtualization platform, one can automate the experiment types for the following categories (c) and (d).

The number of co-located VMs and other resource management-related factors like core affinity or CPU scheduling parameters are part of category (c). We investigate the influence of factors of this category in two different scenarios. The first scenario focuses on the performance impact when increasing the number of co-located VMs (*scalability*), the second focuses on the performance impact when allocating more resources than are actually available (*overcommitment*). For scalability, we stepwise increase the number of VMs in each experiment until all available physical resources are used. For overcommitment, the number of VMs is increased beyond the amount of available resources. The process is illustrated in Figure 5.11. As an example resource type, we use the number of available physical cores c of the machine. In the first case, the number of VMs is increased step-by-step up to c, whereas in the second case the number of VMs is increased in steps of multiples of c. Thereby, we achieve an equal distribution of virtual to physical cores. As an example, to determine the influence of core affinity on scalability and overcommitment, we execute one experiment series as depicted in Figure 5.11 with activated core affinity and one series without using core affinity. In the experiment series with core affinity, each virtual core is automatically pinned to a dedicated physical core such that the virtual cores are distributed equally over all physical cores.

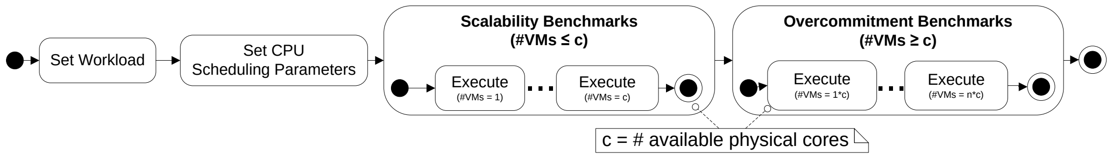

Set Workload
Set CPU Scheduling Parameters
Scalability Benchmarks (#VMs ≤ c)
Execute (#VMs = 1)
Execute (#VMs = c)
Overcommitment Benchmarks (#VMs ≥ c)
Execute (#VMs = 1*c)
Execute (#VMs = n*c)
c = # available physical cores

Figure 5.11.: Benchmark execution in scalability and overcommitment scenarios.

Finally, for category (d) we execute a set of benchmarks focusing on the different types of workloads. Furthermore, we also execute VMs with different workload types in parallel to determine the mutual influences of the workload types.

5.2.2.3. Metrics and Benchmark Selection

To quantify the effects of the performance-influencing factors, one can use different metrics (e.g., response time, throughput, bandwidth, utilization). We decided to use throughput (i.e., the number of completed jobs/operations per time unit) as our metric since we are interested in comparing alternatives at maximum performance. As benchmark candidates for CPU and memory intensive workloads, we considered Passmark PerformanceTest (2009), a benchmark also used by VMware (2007), and SPEC CPU2006 (2006), an industry standard CPU benchmark. For I/O intensive workloads, we considered Iometer (2006) (formerly developed by Intel Corp.) and Iperf (2003), also used by Apparao et al. (2006).

Passmark PerformanceTest is a benchmark focused on CPU and memory performance. The benchmark rating is a weighted average of several single benchmark categories (CPU, memory, I/O, and so on). In the CPU category the properties of the CPU are benchmarked. To this end, several benchmark units e.g., integer or floating point operations are executed. Together, the results of these benchmarks form the CPU Mark rating. The same procedure is applied for the memory benchmark and so on. To get a comparable overall rating, the results of each category are weighted and then added up to the final Passmark rating. SPEC CPU2006 is an industry standard benchmark for evaluating CPU performance. It is structured in a similar fashion and consists of CINT2006 integer benchmarks and CFP2006 floating point benchmarks. The benchmark metrics are calculated from several sub-benchmark results like bzip2 or gcc runs. Both benchmarks have a similar structure consisting of sub-benchmarks to calculate an overall metric. However, unlike Passmark, SPEC CPU2006 does not distinguish between CPU and memory performance and a SPEC CPU2006 benchmark run can take several hours to complete.

For I/O intensive workloads we used the Iometer benchmark which measures the performance of disk and network controllers as well as system-level hard drive performance. Iometer consists of both a workload generator (executes I/O operations in order to stress the system) and a measurement tool (examines the performance of the I/O operations). Furthermore, for network performance measurements, the Iperf benchmark can be used. It is based on a client-server model and supports the throughput measurement of TCP and UDP data connections between both endpoints. It establishes a control connection and a separate data connection for the measurement itself, using the APIs and protocols for the specified test.

Finally, further workloads that can be leveraged as part of our automated experimental analysis are provided by SPEC standard benchmarks such as SPECjbb2005 (stressing CPU and memory performance), SPECmail2009 (stressing I/O performance) and SPECjEnterprise2010 (emulating a complex three tier e-business application). These benchmarks are partly used together with others in the new SPECvirt benchmark which is currently under development. However, we do not consider this benchmark here as it calculates an overall metric to compare servers and different virtualization platforms. It is not designed to analyze the influence of specific factors on the system performance.

5.2.2.4. Experiment Hardware

We conducted experiments in two different hardware environments. Unless stated otherwise, we used Windows 2003 Server as operating system that hosts the benchmark application.

Environment 1: This experiment environment is a standard desktop *HP Compaq dc5750* machine with an Athlon64 dual-core 4600+, 2.4 GHz. It has 4 GB DDR2-5300 of main memory, a 250 GB SATA HDD and a 10/100/1000-BaseT-Ethernet connection. In some of our experiments, we also used this hardware to run experiments on a single core of the CPU by deactivating the second core in the OS.

Environment 2: To be able to evaluate the performance when scaling the number of VMs, we used a more powerful machine, the *SunFire X4440* x64 Server. It has four 2.4 GHz AMD Opteron six-core processors with 3 MB L2, 6 MB L3 cache each, 128 GB DDR2-667 main memory, eight 300 GB serial attached SCSI storage drives and four 10/100/1000-BaseT-Ethernet connections.

5.2.2.5. Experiment Results on Citrix XenServer 5.5

We now present experiment results with the Citrix XenServer 5.5 virtualization platform in the described environments. We chose Citrix XenServer 5.5 as a representative virtualization platform because it is a freely available virtualization platform with a significant market share. It is based on the bare-metal open source hypervisor Xen. With the Xen hypervisor, multiple para-virtualized (DomU Guests) or full-virtualized virtual machines (HVM Guests) can be executed on a single server sharing the physical resources, depicted in Figure 5.12. 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 (Dom0). The control domain contains the device drivers to access the physical devices. All communication of the guest domains with the physical devices goes through Dom0. This causes additional management overhead

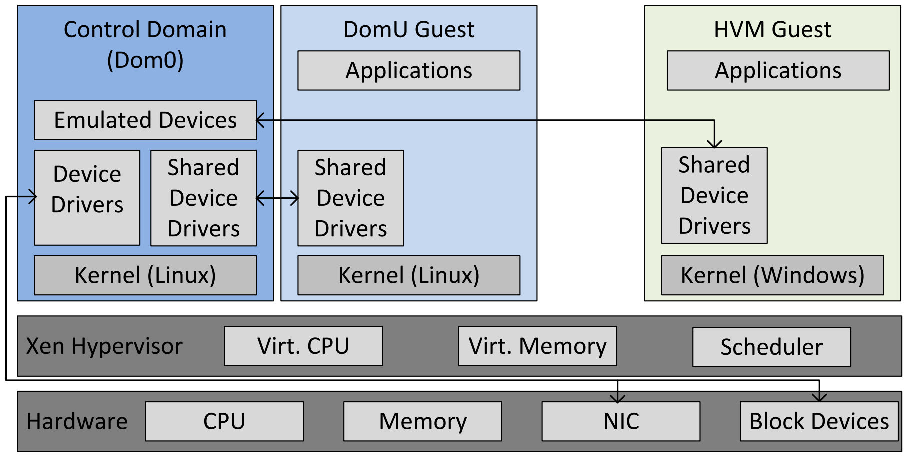


Figure 5.12.: Architecture of the Xen hypervisor.

in terms of CPU consumption. For example, if a guest domain sends a disk I/O request, Dom0 requires CPU time to process the request on behalf of the guest domain.

In the following experiments, we evaluate the influence of the following performance-influencing factors. As *VirtualizationType*, we use full (hardware) virtualization, since XenServer supports this feature and it is the most common type used in practice. As already mentioned, the *VmmArchitecture* of Xen contains a dedicated control domain Dom0. Regarding the *WorkloadProfile*, we investigate CPU, memory and network intensive workloads independently and in combination with each other. Concerning the *ResourceManagementConfiguration*, we investigate the influences of the memory management and the credit-based CPU scheduler implemented in the Xen hypervisor. The latter is influenced by the three parameters core affinity, CPU allocation, CPU priority. We also put a special focus on varying the number of co-located VMs.

We organize our results in four different scenarios. The first scenario summarizes the performance overhead of the Xen hypervisor for CPU, memory, and I/O intensive workloads compared to a native system. The second scenario presents the influences of core affinity when scaling up resources. The third scenario analyzes the influence of the number of VMs on the performance of the virtualization platform in situations when over-committing resources. The fourth scenario compares the mutual performance impact of different workload types.

Virtualization Overhead

In this scenario, our goal is to quantify the performance-influencing factors of the Citrix XenServer 5.5 virtualization platform and compare it to a native system for CPU, memory, and I/O intensive workloads. To this end, we executed Passmark, SPEC CPU2006 and Iperf benchmarks in a native and virtualized setup of our two experiment environments. In the native case, we executed the benchmark directly on the operating system. In the virtualized environment, the benchmark was executed in a single VM. Experiments with more than one VM are part of separate scenarios.

The results of these experiments are depicted in Table 5.2. Note that for the SPEC benchmark results, we can only publish the relative values because of licensing reasons. The data shown includes absolute values measured in the native and virtualized system as well as the absolute and relative delta, whereas the relative delta is the ratio of the absolute delta and the native system performance. For the two CPU intensive benchmarks, the results show a similar, almost negligible performance overhead. In both cases, the

Table 5.2.: CPU, memory and network benchmark ratings on the native and virtualized HP Compaq dc5750.

CPU Benchmark Ratings	**native**	**virtualized**	**delta (abs.)**	**delta (rel.)**
Passmark CPU, 1 core	639.3	634.0	5.3	0.83%
Passmark CPU, 2 cores	1232.0	1223.0	9.0	0.97%
SPECint®_base2006				3.61%
SPECfp®_base2006				3.15%

Memory Benchmark Ratings	**native**	**virtualized**	**delta (abs.)**	**delta (rel.)**
Passmark Memory, 1 core	492.9	297.0	195.9	39.74%
Passmark Memory, 2 cores	501.7	317.5	184.2	36.72%

Network Benchmark Ratings	**native**	**virtualized**	**delta (abs.)**	**delta (rel.)**
Iperf, send	527.0	393.0	134.0	25.43%
Iperf, receive	528.0	370.0	158.0	29.92%

Disk Benchmark Ratings	**native**	**virtualized**	**delta (abs.)**	**delta (rel.)**
Passmark Disk	572.40	407.45	164.95	28.82%

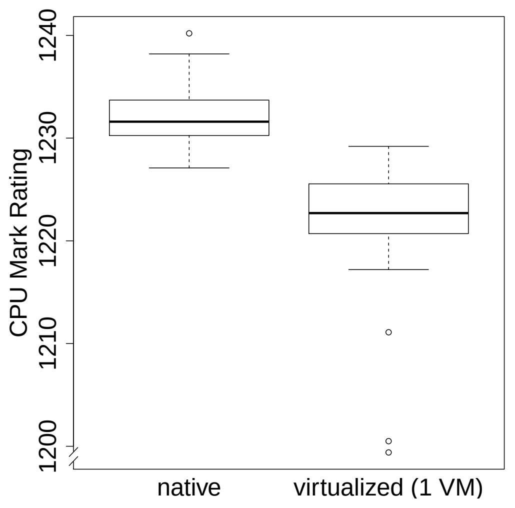
performance degradation when switching from a native to a virtualized system remains below 4%.

When comparing the performance of a memory-intensive workload (Table 5.2, Figure 5.13b), one can observe a much higher performance degradation in the virtualized system (about 40%). The reason for the discrepancy between CPU and memory benchmark results is the fact that at the time of our experiments, CPU virtualization was well-understood and supported by the hardware, whereas memory virtualization was still rather immature and had no hardware support (Rosenblum and Garfinkel, 2005).

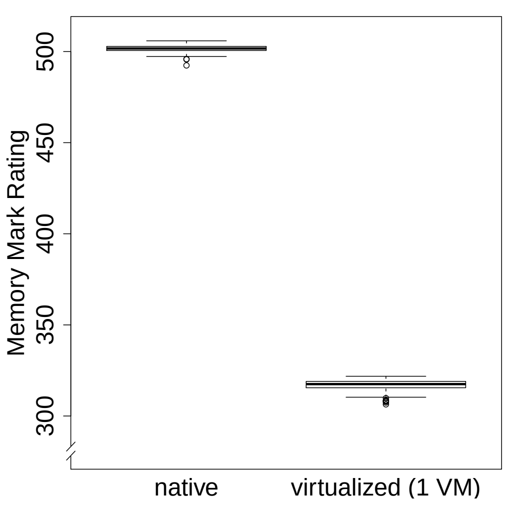
Table 5.2 and Figure 5.13c depict the results of the network performance measurements with Iperf. In our experiment setup, the client and server were connected with a DLink 1 Gbit switch. We observe a performance decrease for TCP connections in both directions, 25% upstream (Client to Server) and 30% downstream (Server to Client). This shows that like for memory virtualization, there is still a relatively high performance loss because of the missing hardware support. Also interesting is that performance degradation of sending and receiving network traffic differs by 5%. Furthermore, also the disk I/O benchmark ratings indicate a high performance loss of 29%.

To gain a better understanding of the exact cause of the overheads, we investigated the performance degradation for CPU, memory and I/O in more detail by looking at the more fine-grained Passmark PerformanceTest sub-benchmark results. Figure 5.14a depicts the Passmark CPU Mark sub-benchmark results, normalized to their native execution. These results demonstrate that floating point operations are more expensive (up to 20% performance drop for the `Physics` sub-benchmark) than other operations.

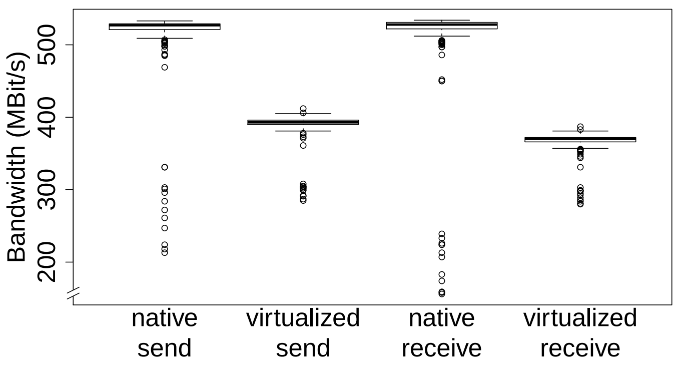
Furthermore, looking at the fine-grained memory sub-benchmark results depicted in Figure 5.14b, one can see that for the memory-intensive workloads the main cause for the overall performance drop stems from the allocation of large memory areas. For the `Large RAM` sub-benchmark, performance overhead is almost 97%. The problem was that to replicate our VM template in the CPU overcommitment scenarios, we could only assign 256 MB main memory to each VM because memory overcommitment is currently not supported by Citrix XenServer 5.5. We confirmed this in a separate, independent experiment with a single VM with 3 GB of main memory. In this experiment, we observed that the performance overhead for large memory accesses is only 65% instead of 97%, which also improves the overall memory benchmark results slightly. Hence, increasing memory allocation can



(a) CPU



(b) Memory



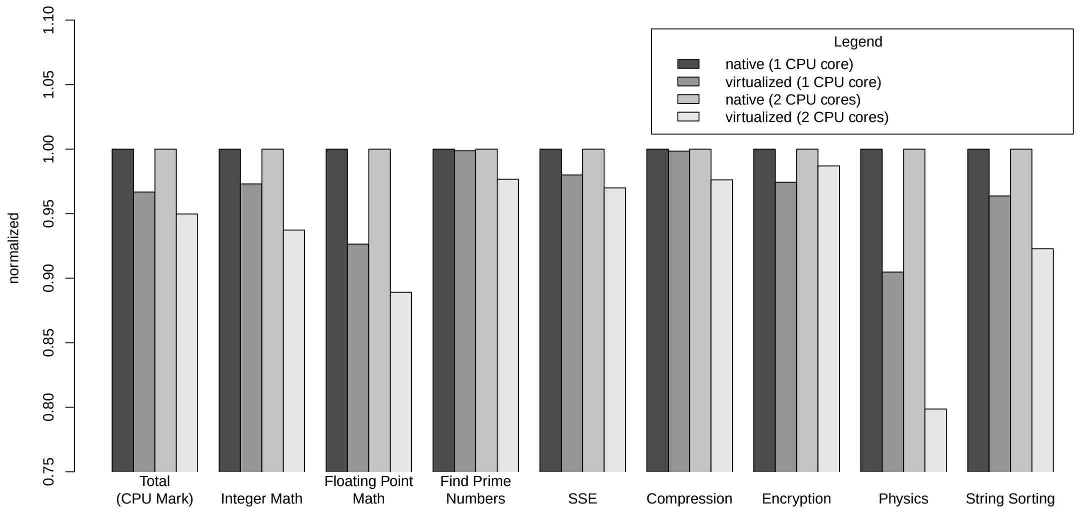
(c) Network

Figure 5.13.: Native vs. virtualized Passmark CPU and memory results and the Iperf benchmark results on the HP Compaq 5750.

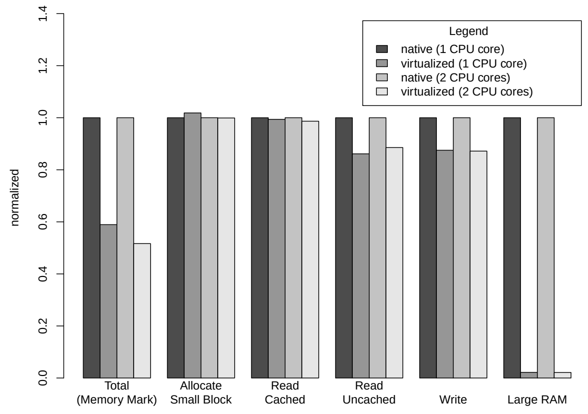
significantly improve performance for memory-intensive workloads, especially if expensive swapping can be avoided.

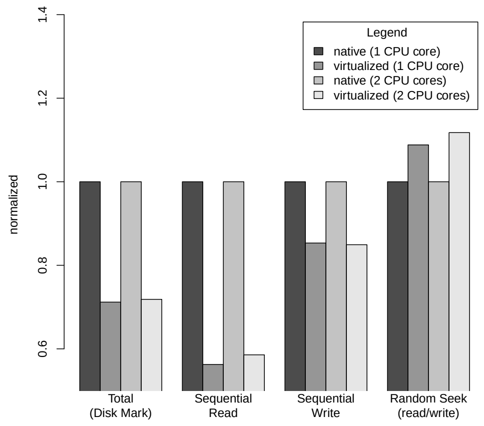
To examine the virtualization overhead for network I/O in more detail, we conducted further experiments with the Iperf benchmark. The goal of these experiments was to examine how the additional overhead introduced by the hypervisor to handle I/O requests is distributed among the running VMs. To this end, we executed four VMs, two VMs running CPU Mark and two VMs running Iperf. We pinned them pairwise on two physical cores. Core zero ($core_0$), where the Dom0 is executed, executed a pair of the CPU VM and the Iperf VM and a different available core ($core_x$) executed the other pair. The CPU benchmark was executed on both VMs, simultaneously, and the network I/O benchmark was started separately on one VM. This symmetric setup allows us to compare the results of VMs executed on $core_0$ with the results on the different $core_x$.

One would expect that there is no performance effect on the VMs running on $core_0$ when the Iperf VM on $core_x$ executes network I/O. However, the results show that the performance of the VM running the CPU benchmark on $core_0$ drops up to 13% when the VM on $core_x$ is executing the network I/O benchmark. Figure 5.15 depicts the CPU benchmark results of VMs executed on $core_0$ ($\circ$) and $core_x$ (+). The second VM on $core_x$ receives the network load, hence the benchmark rating of the VM sharing this core drops significantly.



(a) Passmark CPU Mark metrics





(b) Passmark Memory Mark metrics

(c) Passmark Disk Mark metrics

Figure 5.14.: Sub-benchmark results of the Passmark CPU, Memory, and Disk Mark.

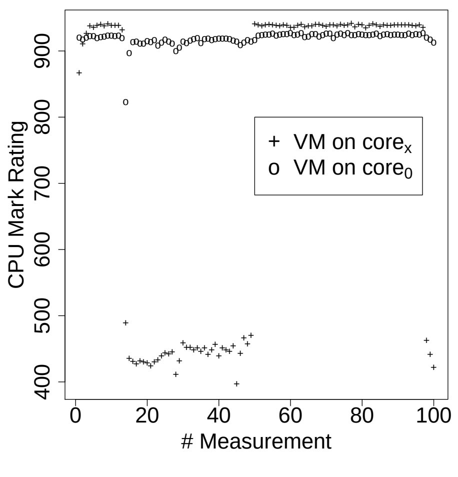


Figure 5.15.: CPU benchmark results of two VMs executed on $core_0$ and $core_x$. Network traffic is received by the VM sharing $core_x$.

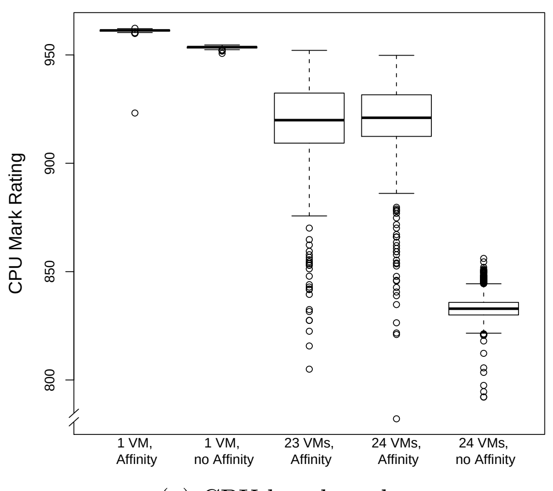
But also the benchmark rating of the VM on $core_0$ drops, although its paired VM is idle. Because VMs executed on other cores than $core_0$ did not exhibit this behavior, this indicates that Dom0 mainly uses $core_0$ to handle I/O. This causes a slight performance drop for VMs simultaneously executed on $core_0$, i.e., about 1% on average. However, this drop could further increase if further machines on other cores receive network load.

Figure 5.14c shows our detailed sub-benchmark results for disk I/O intensive workloads. With the Passmark Disk Mark benchmark, we measured a performance overhead of up to 28%. A more detailed look at the benchmark results shows that most of the performance overhead is caused by sequential `read` requests, which achieve only 60% of the native performance, whereas for `write` requests, the performance overhead remains below 20%. The performance speed-up for the random seek benchmark can be explained by the structure of the virtual block device, a concept used in Citrix XenServer 5.5 for block-oriented read and write operations, minimizing the administration overhead and thus decreasing access times.

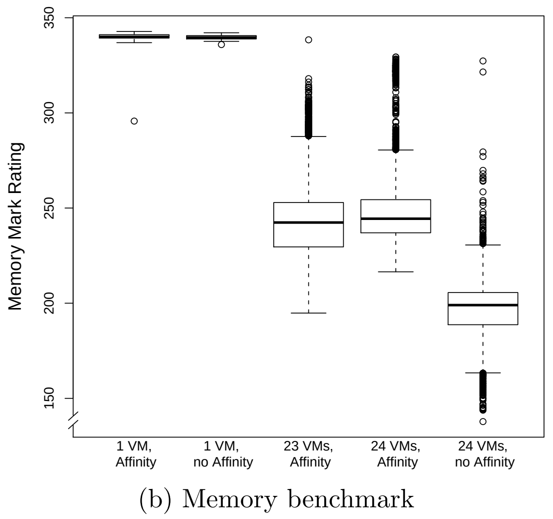
In summary, we can say that for full virtualization the performance overhead of CPU, memory and I/O virtualization amounts to 5%, 40%, 30%, respectively.

Scalability and Core Affinity

In a virtualized environment, the scheduling of virtual resources to the physical resources has a significant influence on the VM performance (Apparao et al., 2008). For example, imagine a machine with 24 cores, each core having its own cache. If a VM is re-scheduled from one core to another, its performance will suffer from cache misses because the benefit of a warm cache is lost. To avoid this, current virtualization platforms provide means to assign cores to VMs, called *core affinity* or *core pinning*. With core affinity, the VM is executed only on the assigned core(s) which in turn has a significant influence on the cache and core contention and hence on performance. In this scenario, we quantify the performance influence of core affinity. Additionally, we investigate the performance overhead when scaling the number of concurrent VMs up to the limit of available resources, i.e., CPU cores.



(a) CPU benchmark



(b) Memory benchmark

Figure 5.16.: Performance influence of core affinity on CPU and memory benchmarks on the SunFire X4440. The box plots depict aggregated benchmark results for all VMs in the experiment.

We tested the effect of core affinity using several experiments on the 24-core SunFire X4440 (see Table 5.3 and Figure 5.16). When comparing the CPU and memory benchmark rating of one VM running with no affinity and one VM pinned to a dedicated core, the performance changes about 0.80% for the CPU benchmark and 0.10% for the memory benchmark. Hence, for one VM there is no significant performance influence when activating affinity. However, when comparing the benchmark results of the same experiment for 24 VMs (each VM has one virtual CPU), performance increases with affinity by 88.1 (9.56%) and 46 (18.82%) for the CPU and memory benchmark, respectively.

Table 5.3.: CPU and memory benchmark results for different core affinity experiments. The values shown are the median over all benchmark runs (200 for one VM, $200 \cdot 24$ for 24 VMs) on the SunFire X4440.

	CPU Mark			**Memory Mark**		
	no affinity	affinity	rel. delta	no affinity	affinity	rel. delta
1 VM	953.60	961.30	0.80%	339.95	339.60	0.10%
24 VMs	832.90	921.00	9.56%	198.40	244.40	18.82%
rel. delta	12.66%	4.19%	-	41.64%	28.03%	-

The performance loss of one VM without affinity compared to 24 VMs without affinity is 120.7 (12.66%) and 141.55 (41.64%) for the CPU and memory benchmark, respectively. However, when increasing the amount of VMs from one VM with core affinity to 24 VMs with core affinity, performance drops only by 40.3 (4.19%) and 95.2 (28.03%), respectively. Hence, on average, 8.47% of the performance penalty for the CPU benchmark and 13.61% for the memory benchmark can be avoided when using core pinning. Also interesting observations are that the variability of the benchmark results increases with the number of co-located VMs, which can lead to performance degradations of up to 10% in the worst case.

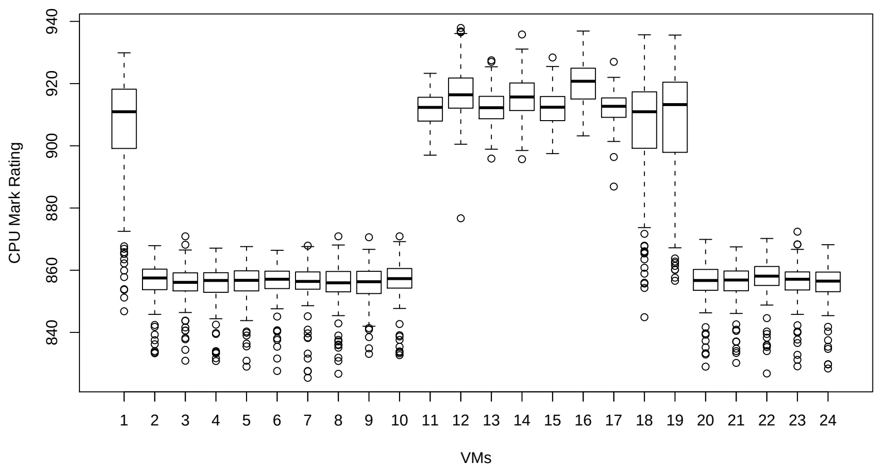
Another interesting fact observable in Figure 5.16 is that there is little difference in performance between 23 VMs and 24 VMs, both with affinity. In the case of 23 VMs, one core is free and can be used by the hypervisor. However, the median of both measurements deviates only by 0.12% for the CPU benchmark and 0.83% for the memory benchmark. Hence, leaving one core for the hypervisor has no significant effect on reducing the performance degradation introduced by virtualization.

To gain further insight and explain why core affinity improves performance, we compared the benchmark ratings of the individual VMs in settings with and without core affinity. Figure 5.17a shows the box plot of 24 VMs simultaneously executing the CPU benchmark without core affinity. There are clearly two categories of VMs, one category (1, 11–19) performing significantly different from the other (2–10, 20–24). This behavior is not observed in a scenario with core affinity (see Figure 5.17b), when each VM is executed on a separate physical core. This indicates that XenServer's scheduler does not distribute the VMs on the cores, equally. In general, it demonstrates that multi-core environments are still a challenge for hypervisor schedulers. In Section 5.4.2, we investigate if this effect is observable on other hypervisor architectures, too.

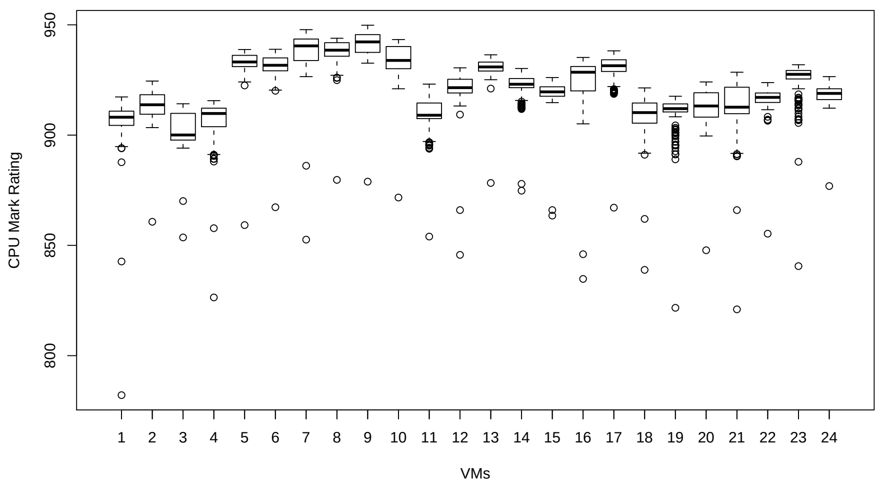
From the above results we conclude that core affinity has a significant effect on virtualization performance. Performance can gain up to 20%, depending on the ratio of executed virtual machines and available resources.

Overcommitment

In this scenario, we investigate the performance degradation when systematically overcommitting resources. Overcommitment in this context means that the resources issued to all VMs in total exceed the actually available physical resources. For example, if we have a dual core machine running four VMs with one virtual CPU each, we have over-committed the available physical CPU resources by a factor of two.



(a) without core affinity



(b) with core affinity

Figure 5.17.: CPU benchmark results for 24 VMs executed without and with core affinity.

In the following experiments, we scaled the amount of VMs (each VM is configured with one virtual CPU) from one up to four times the amount of physically available CPUs (overcommitment level of $x \in \{1, \ldots, 4\}$). In case of the HP Compaq dc5750 environment, we increased the number of VMs to 2, 4, 6 and 8 and for the SunFire X4440 to 24, 48, 72 and 96. We also created a single core scenario with the SunFire X4440 in which we deactivated all but one physical core and increased the number of VMs to 1, 2, 3 and 4. The trend of the absolute results for the CPU and memory benchmark in the SunFire X4440 environment is depicted in Figure 5.18.

Figure 5.19 compares the normalized CPU rating of both hardware environments and the single core scenario. We observe that performance decreases roughly about $1/x$. Interestingly, for the CPU benchmark, the measured performance is slightly better than this expected theoretical value. The reason for this observation is that the execution of a single benchmark instance cannot utilize the CPU at completely 100%. Hence, there are unused CPU cycles which are utilized when executing multiple VMs in parallel. When increasing the number of VMs up to 72, we observed a CPU utilization of all cores at 100%. This

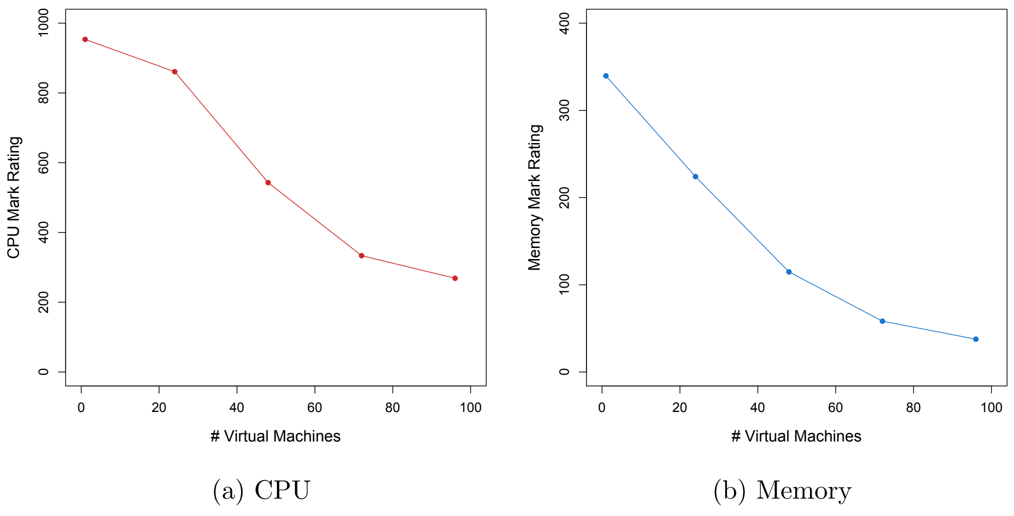


Figure 5.18.: Absolute scalability and overcommitment experiment results of the CPU and memory benchmark on SunFire X4440.

effect, however, does not apply to the memory-intensive workload. Therefore, the memory benchmark rating is slightly worse than the expected theoretical value. Finally, we observe that the single core performs better than the HP Compaq which in turn is better than the SunFire. This indicates that more cores cause more scheduling overhead, causing higher performance degradation. Intuitively, one might also assume that performance decreases faster or even suddenly drops when over-committing system resources. However, the results show that the performance degradation can be approximated by $1/x$, which is the optimal theoretical value. Moreover, one can see that the performance degradation is very similar in both hardware environments (max. 10% deviation). This is remarkable because one would intuitively assume that the SunFire would perform significantly better because it has more physical resources. To evaluate performance isolation and the fairness of the resource sharing, we depicted CPU and memory benchmark measurements box plots over all 96 VMs in Figure 5.20. They show a low scattering around the median, which indicates a fair resource sharing and good performance isolation.

As a result, we conclude that the performance degradation from over-committing CPU resources by increasing the number of VMs is proportional to the overcommitment factor with an upper limit of $1/x$. Furthermore, we observed that the hardware environment has almost no influence on the scalability and performance degradation and both CPU and memory workloads are affected in a similar way.

Mutual Influences of Workload Types

The goal of the experiments presented next is to identify the mutual influences of VMs sharing their resources while serving different workload types. To this end, we pinned two VMs VM_A and VM_B on the same physical core other then $core_0$ to avoid interferences with Dom0. Then, we ran an experiment for each possible combination of benchmark types. As a result, we calculate the relative performance drop as $r = 1 - (r_i/r_s)$, where r_i is the interference result and r_s the result measured when executing the benchmark on an isolated VM. Table 5.4 summarizes the results for all combinations of workload types. Note that we did not run network vs. network experiments because a different hardware environment with additional network interface cards would have been required.

The results show that there are no significant mutual influences between CPU and memory

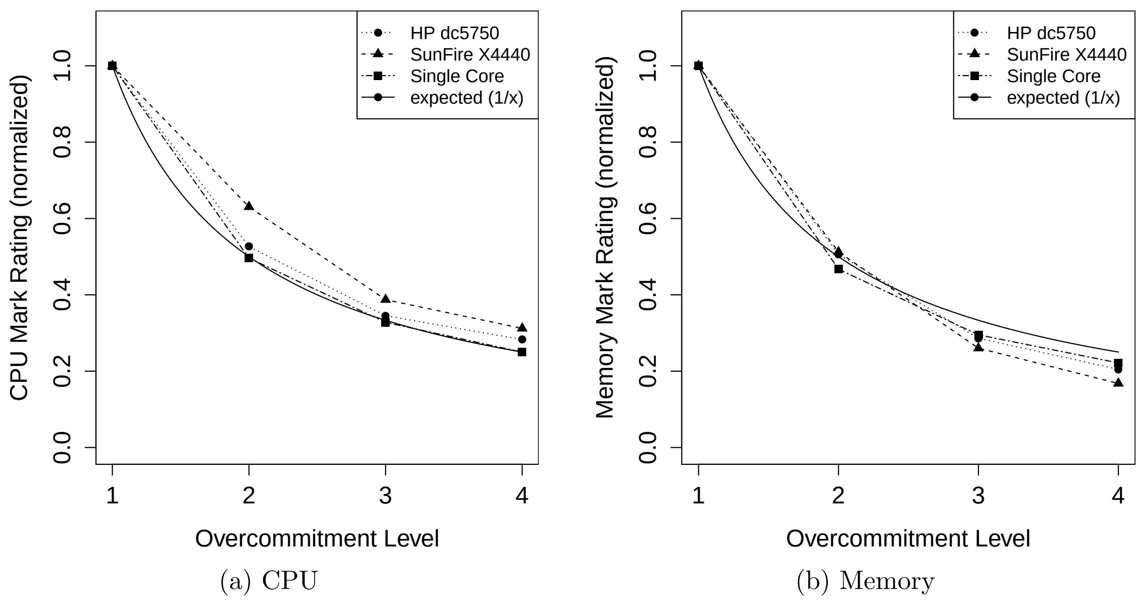


Figure 5.19.: Normalized scalability comparison for CPU and memory benchmarks in the HP dc5750, SunFire X4440 and single core scenarios in comparison to the expected value of $1/x$.

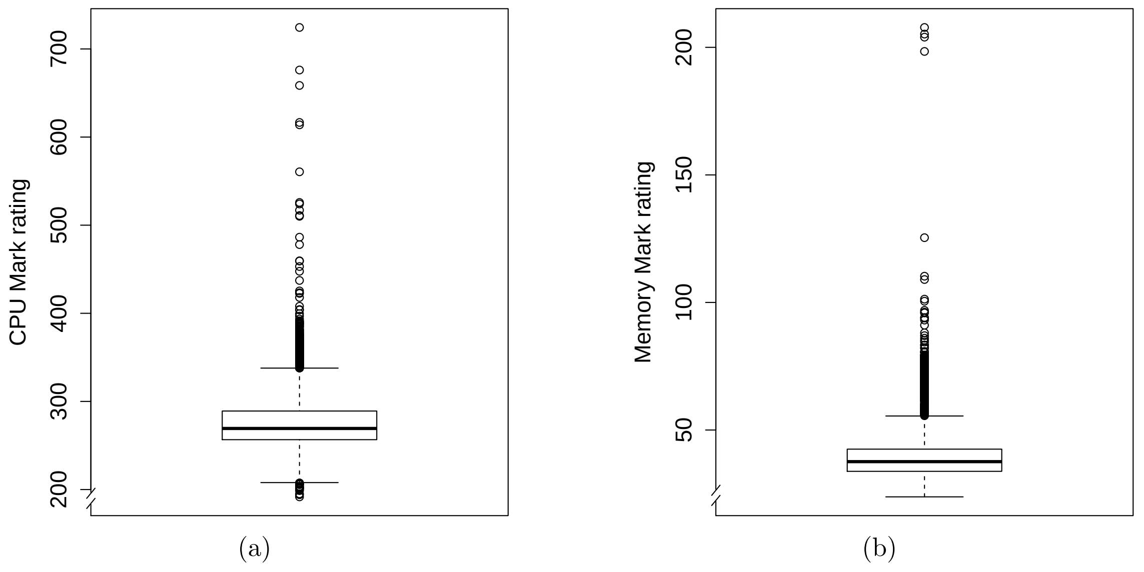


Figure 5.20.: Box plots of the CPU and memory benchmark results over all 96 VMs (overcommitment factor of $x = 4$).

Table 5.4.: Mutual performance degradation for different workload types on Citrix XenServer 5.5.

VM_A	CPU	CPU	Mem	CPU	Mem	Disk	CPU	Mem	Disk
VM_B	CPU	Mem	Mem	Disk	Disk	Disk	Net	Net	Net
r_A	46.71%	50.64%	50.33%	23.35%	24.82%	31.16%	52.88%	52.85%	3.07%
r_B	52.44%	45.93%	49.04%	1.49%	-0.09%	45.99%	40.46%	42.18%	33.31%

intensive workloads. The performance drop for both benchmarks is similar and the drop also fits the expectation that each VM receives only half of its performance compared to an isolated execution. An explanation for this is the similarity of both workload types in terms of the used resources as well as the hardware support for CPU virtualization. An interesting observation is that the disk benchmark is not influenced by other workload types except when executed with another instance of the disk benchmark. This indicates that on Citrix XenServer 5.5, disk intensive workloads do not compete for resources of CPU and memory intensive workloads. This can be explained with a similar reason as for the virtualization overhead of Disk Mark. The concept used in Citrix XenServer 5.5 for block-oriented read and write minimizes the administration overhead because the disk workload can be passed through without requiring major hypervisor intervention.

5.2.3. Derivation of the Performance Model

Based on the experiment results of Section 5.2.2.5, we now propose an analytic performance prediction model to predict the performance impact of virtualization. As explained in Section 5.2.2.3, we measured performance as the amount of benchmark operations processed per unit of time, i.e., the throughput of the system under test. In the following, we calculate a performance overhead factor o which can be used to predict the performance of a virtualized application $p_{virtualized} = o \cdot p_{native}$, where o can be replaced by one of the formulas of the following sections. To evaluate our performance model, we conducted further experiments with VMware ESX 4.0. These results are presented in Section 5.4.2.

Overhead of Virtualization

The following equations allow to predict the overhead that is introduced when migrating a native application to a virtualized platform. These equations assume that there are no influences by other virtual machines. We consider this case separately in the scalability and overcommitment scenarios.

For CPU and memory virtualization, we calculate the overhead factors

$$o_{\{cpu|mem|io\}} = 1 - \frac{relative\_delta_{\{cpu|mem|io\}}}{100}$$

using the measured relative deltas in Table 5.2. For the CPU overhead, our results have shown only an insignificant deviation between Passmark PerformanceTest and SPEC CPU2006. Therefore, and because CPU virtualization is already very mature and hardware supported, we are confident that our measurement results can be generalized. If specific overhead factors are required, one can use our automated approach to determine these factors for any other virtualization platform. For memory and I/O overhead, we recommend to measure the performance overhead for each specific virtualization platform using our automated approach because our experiments showed significant differences between the different virtualization platforms and their implementations, respectively.

Scalability

To model the performance-influence when scaling up CPU resources, we use linear equations derived with linear regression. We define the performance overhead as

$$o_{scal} = a + b \cdot c_{virt}$$

where c_{virt} is the number of virtual cores. The coefficients a and b, derived with linear regression from the experiments in Section 5.2.2.5, are given in Table 5.5. We distinguish between scenarios without core affinity and scenarios, where the virtual CPUs are pinned to the physical cores in an equal distribution. These equations give an approximation of the performance degradation when scaling up, independent of the virtualization platform. However, this approximation is only valid until you reach the amount of physical cores available. The overcommitment scenario is modeled in the next section.

Table 5.5.: Coefficients a, b for the linear equations for CPU and memory performance when scaling-up and the corresponding coefficient of determination.

Scenario	**a**	**b**	R^2
CPU	1.008	-0.0055	0.9957
Memory	1.007	-0.0179	0.9924
CPU (with core affinity)	1.003	-0.0018	0.7851
Memory (with core affinity)	1.002	-0.0120	0.9842

Overcommitment

In situations with over-committed CPU resources, we can approximate the performance overhead as

$$o_{overc} = \frac{c_{phy}}{c_{virt}}$$

where x is the overcommitment factor. The overcommitment factor is determined as the ratio of the available physical resources c_{phy} (here CPU cores), and the provisioned virtual resources c_{virt}. For CPU overcommitment, this dependency between the performance overhead and the overcommitment factor is independent of the virtualization platform and the amount of executed VMs. Our experiments demonstrated that the performance overhead simply depends on the ratio of virtual and physical cores. This dependency is valid at the core level, i.e., if you pin two VMs with one virtual core each on a single physical core, you experience the same performance drop.

This performance model derived for the virtualization platform Citrix XenServer 5.5 can now be used as a CustomResourceConfigurationModel to describe the performance-relevant properties of the virtualization resource layer. This description can then be used in the performance analysis process to increase prediction accuracy. To assess if the derived performance model can be applied for other virtualization platforms as well, we conducted an evaluation with VMware ESX 4.0, presented in Section 5.4.

5.3. Application Architecture, Usage Profile, and Deployment Meta-Models

To conduct performance predictions on the model level, further aspects than just the details about resource landscape must be modeled. These additional aspects are captured in the application architecture meta-model, deployment meta-model, and usage profile meta-model, which are also part of DML (cf. Section 4.2). Although the application architecture meta-model and the usage profile meta-model are not in the focus of this thesis, we briefly explain their main concepts as they are part of the overall model-based adaptation approach.

5.3.1. Application Architecture Meta-Model

The application architecture is modeled after the principles of component-based software systems. Szyperski et al. (2002) defined a software component as a unit of composition with explicitly defined provided and required interfaces. For convenience, we also use the term *service* to refer to a method signature of a software component's interface. Details about the meta-model specification of interfaces can be found in Section A.1.

A component may be either a *basic* (i.e., atomic) component or a *composite* component. A composite component may contain several child component instances, assembled through so-called *assembly connectors*, connecting required interfaces with provided interfaces. A component-based *system* is modeled as a special type of composite component that provides at least one interface. An example of how a composite component is assembled is shown in Figure 5.21. Component *ComponentA* comprises three instances of basic components. By assembling components to build a composite component or a system, they are "instantiated" and each of them is then referred to as *assembly context*. Assembly contexts form the part of the system that can be deployed on the containers of the resource landscape (see Section 5.3.3 for more details).

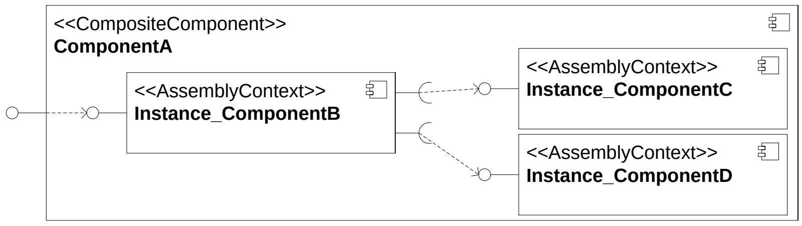


Figure 5.21.: Assembly of a composite component.

To describe the performance behavior of a service offered by a component, the application architecture meta-model supports multiple (possibly co-existing) behavior abstractions at different levels of granularity (cf. Section A.1 for further details). The behavior descriptions range from a *black-box* abstraction (a probabilistic representation of the service response time behavior), over a *coarse-grained* representation (capturing the service behavior as observed from the outside at the component boundaries, e.g., frequencies of external service calls and amount of consumed resources), to a *fine-grained* representation (capturing the service's internal control flow and covering performance-relevant actions). The multiple abstraction levels make it possible to use the model in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to detailed system simulation. Moreover, one can choose the modeled abstraction level depending on the information that monitoring tools can obtain at run-time, e.g., to what extent component-internal information is available.

Furthermore, the behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation. The application architecture meta-model provides modeling abstractions and concepts for expressing and resolving such parameter and context dependencies. For example, probabilistic characterizations of parameter dependencies that are based on monitoring data are supported. Further details of the application architecture meta-model can be found in Section A.1. The complete specification of the application architecture model is part of the work of Brosig (2014).

5.3.2. Usage Profile Meta-Model

To model user interactions with the system (i.e., the usage profile), DML provides a usage profile meta-model. A usage profile contains one or more usage scenarios, which can be seen as a combination of UML use cases and UML activities. A usage scenario describes the workload type (e.g., open or closed workload), the workload intensity (e.g., request arrival rates), and the user behavior, i.e., which services are called and in what sequence. It can be used to describe, e.g., which software components of the system are directly invoked by users and in what specific usage scenarios. Both workload type and scenario behavior can contain additional stochastic information to characterize, e.g., the probabilities of choosing alternative services or the number of users in the system. The usage profile meta-model is part of the work of Brosig (2014) and the meta-model specification can be found in Section A.2.

5.3.3. Deployment Meta-Model

To capture the relationship between the resource landscape and the application architecture, one must model the connection between hardware and software. This connection can be described with the deployment meta-model depicted in Figure 5.22. This model allocates software component instances of the application architecture meta-model on container instances of the resource landscape meta-model.

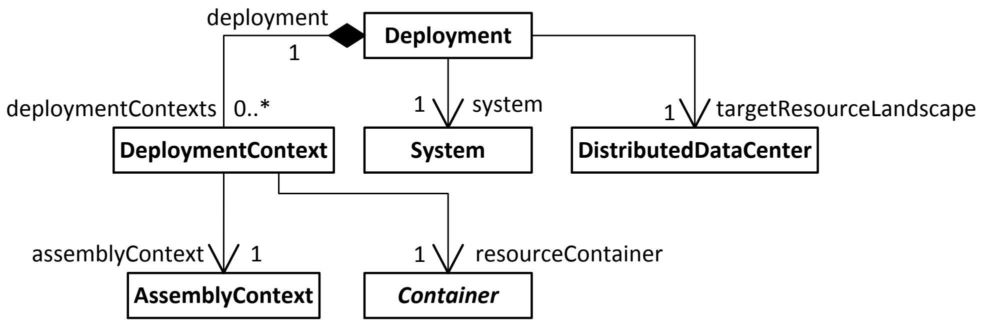


Figure 5.22.: The deployment meta-model.

A Deployment has a reference to a DistributedDataCenter (the root element of the resource landscape meta-model) and a System, the root element of the application architecture meta-model. Note that each element of the application architecture that is deployable (e.g., basic component, composite component, or a subsystem) is modeled as an Assembly-Context. The instantiation of two different components of the same type is modeled using two different AssemblyContexts. The actual connection between assembly context instances of the application architecture and container instances of the resource landscape is modeled using DeploymentContexts, i.e., a DeploymentContext is a mapping of an AssemblyContext to a Container.

5.4. Case Studies

In the following sections, we present two different case studies. In the first case study, we illustrate the advantages of our resource landscape meta-model in a VM (re-)deployment scenario. In the second case study, we apply our approach for automated experimental analysis of performance-influencing factors to the virtualization platform VMware ESX 4.0, to evaluate the previously presented performance overheads measured for Citrix XenServer 5.5.

5.4.1. Modeling Data Centers with the Resource Landscape Meta-Model

In this section, we illustrate the novel concepts of the resource landscape meta-model. To this end, we use an example model instance of the cluster environment in our local data center. We analyze the model instance using simulation and leverage the modeled resource landscape information to improve system adaptation and resource management.

Our example data center consists of six compute nodes from our local cluster environment (see Figure 5.23). Each compute node is equipped with two Intel Xeon E5430 quad-core

CPUs running at 2.66 GHz and 32 GB of main memory. The machines are connected by a 1 GBit LAN. On five of these compute nodes we run XenServer 5.5 as a virtualization platform. The VMs are initially equipped with eight virtual CPUs (each vCPU corresponds to a physical core, i.e., no over-commitment). The sixth compute node is not virtualized and hosts the database.

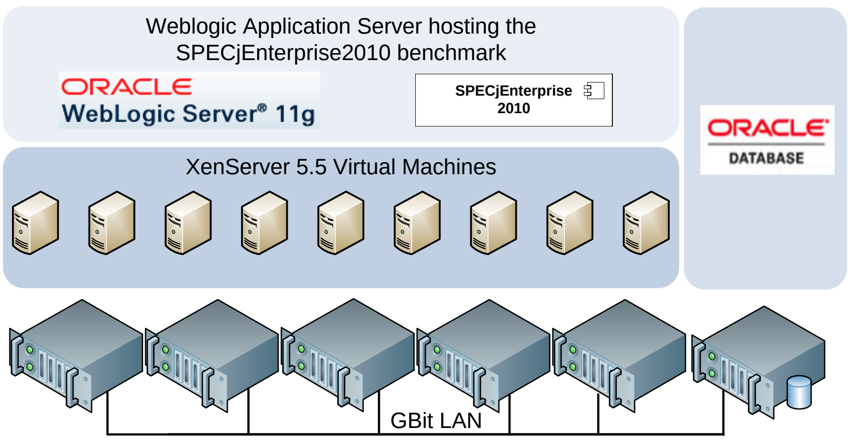

Weblogic Application Server hosting the SPECjEnterprise2010 benchmark
ORACLE WebLogic Server® 11g
SPECjEnterprise 2010
XenServer 5.5 Virtual Machines
ORACLE DATABASE
GBit LAN

Figure 5.23.: Experiment environment consisting of six virtualized cluster nodes and a native database server.

On top of this infrastructure and inside the VMs, we execute the SPECjEnterprise2010 benchmark[1], a representative, state-of-the-art application we have modeled and used in the context of automated model extraction (Brosig et al., 2011) and dynamic resource allocation (Huber et al., 2011a). For this evaluation, the VMs run SPECjEnterprise2010 instances that belong to different customers and each customer has its own performance requirements, specified as SLAs. On the one hand, it is required to maintain these SLAs, the system must be able to scale and provide enough resources in situations where, e.g., the workload varies. On the other hand, it is required that the resources of the data center are used as efficiently as possible, i.e., we have to find a trade-off between these requirements.

Figure 5.24 shows the resource landscape model instance of our data center and the configuration of the resource containers. It reflects the structure of the resource containers. The performance-relevant resource configuration specification of the container templates is given in Figure 5.6. The root element is our local cluster environment called *AcamarCluster*. It groups all contained *ComputeNodes*. Such structural information can be beneficial for system reconfiguration, e.g., for VM migration. Migrating a VM may be restricted for technical reasons, e.g., the NFS share is only accessible for all compute nodes within the cluster. Each compute node refers to the *ComputeNodeTemplate* which specifies the resource configuration of the respective compute node. Similarly, the nested *XenServer* and *VM* runtime environments refer to their type-specific templates. Note that *ComputeNode6* (*Cn6*) contains no hypervisor runtime environment as it is a native system. On top of these containers, we deployed the SPECjEnterprise2010 component instances of the different customers.

This example illustrates the following benefits of our meta-model. First of all, with the template mechanism, it is possible to have multiple instances of the same type in the model instance. If one changes the configuration of the template, all instances in the model referring to the changed template are affected by the change. However, with the override

[1]SPECjEnterprise2010 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 results or findings in this thesis 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`.

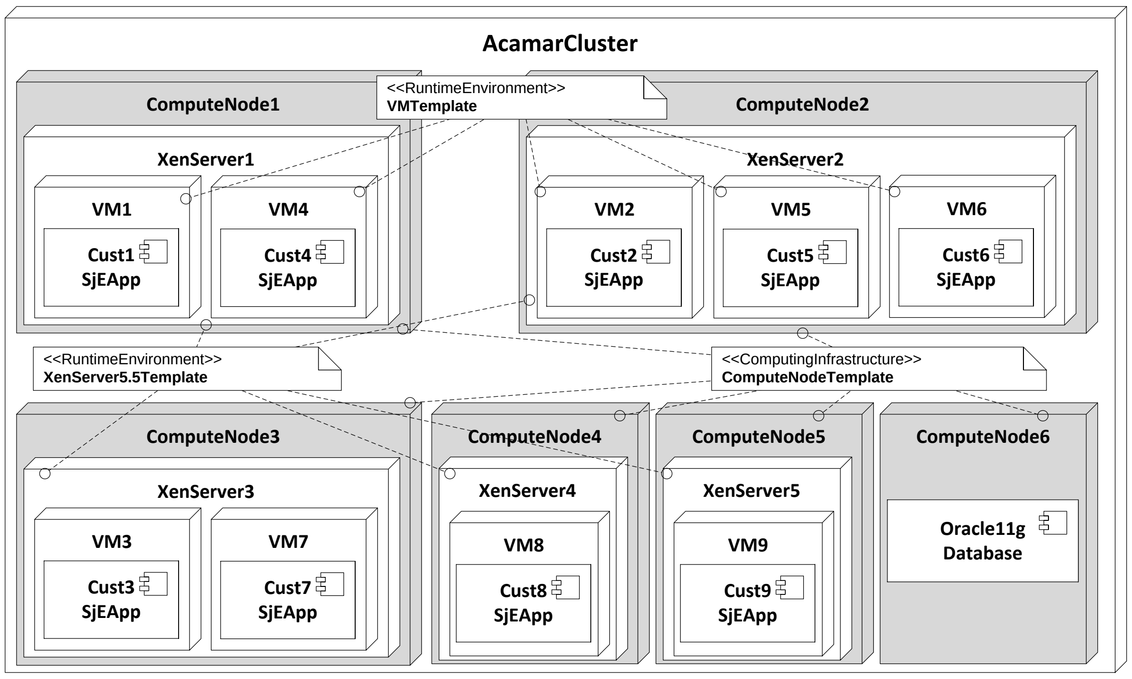


Figure 5.24.: Initial deployment (Depl_0) of the SPECjEnterprise2010 benchmark customer instances.

mechanism as described in Section 5.1.1, it is also possible to have individual specifications for each instance of, e.g., a VM. To undo the individual specification of a container, one can simply delete this specific property to fall back to the template configuration.

To show the benefits of the novel concepts of the resource landscape meta-model for autonomic system adaptation and resource management, we conduct an experiment illustrating how the meta-model and its analysis results can improve adaptation and resource management decisions. To this end, we use the described model instance for run-time (re-)deployment of virtual machines. The performance predictions are obtained using simulation of the model instance.

Table 5.6.: Utilization results for different workload intensities and deployment options.

Compute Node (VM) Utilization [in %]	Deployment Scenario (Workload)				
	Depl_0 (Default)	Depl_0 (High)	Depl_1 (High)	Depl_2 (High)	Depl_3 (High)
Cn1 = VM1(+VM4)	59.5=33.6(+25.9)	92.9=67.2(+25.7)	67.4	66.9	67.4
Cn2 = VM2+VM5+VM6(+VM4)	45.6=25.5+9.6+9.9	45.0=25.5+9.6+9.9	70.9=25.8+9.9+9.6(+25.6)	45.2=25.6+9.9+9.7	45.3=25.6+10.0+9.7
Cn3 = VM3+VM7(+VM4)	62.5=39.8+22.7	61.8=39.2+22.6	62.1=40.0+22.1	88.0=38.5+22.6(+25.9)	61.8=39.4+22.4
Cn4 = VM8(+VM4)	58.2	58.6	58.3	59.0	83.9=58.3(+25.6)
Cn5 = VM9	67.3	67.5	67.8	68.3	67.8
Cn6 = DB	10.5	12.2	12.3	12.3	12.2

(Re-)Deployment of Virtual Machines

We start with a scenario consisting of nine independent instances of the SPECjEnterprise2010 benchmark, each instance for a separate customer. The initial deployment (referred to as Depl_0) of these benchmark instances in the data center is depicted in Figure 5.24. Assume that Cust2, Cust6 and Cust9 are gold customers and their SLAs guarantee an average response time below 20 ms for the default workload, whereas Cust1, Cust3, Cust4, Cust5 and Cust8 are silver customers with guaranteed response times below 40 ms. Another constraint is that the utilization of all compute nodes must be below 90% to avoid heavy response time fluctuations at high system load. In this initial deployment Depl_0 depicted in Figure 5.24 and with the default workload (Default), the requirements are fulfilled, i.e., the system is in a valid state (see column `Depl_0 (Default)` in Table 5.6).

Now, assume that the workload of `Cust1` doubles. The simulation results shown in column `Depl_0 (High)` of Table 5.6 for the increased workload show that the utilization of the compute node *Cn1* would be above the limit of 90%, requiring a re-deployment of the VMs on *Cn1*. Intuitively, one would at first try to migrate the VM with the higher load to the part of the system with the least utilized resources (e.g., *Cn6* or *Cn2*). However, we will see that querying the model and using its architectural information leads to different results.

Table 5.7.: Simulated response times for different deployment options.

Scenario (Workload)	**Response Time [ms]**		
	`Cust2`	`Cust3`	`Cust8`
`Depl_0 (High)`	17.10	19.84	26.55
`Depl_1 (High)`	23.70	19.99	26.65
`Depl_2 (High)`	17.02	31.88	26.43
`Depl_3 (High)`	16.95	19.68	62.00

The model and its analysis results show that *Cn6* has plenty of resources. However, migrating any of the VMs to *Cn6* is not an option because *Cn6* is not virtualized, i.e., it has no runtime environment HYPERVISOR which could host an OS_VM. Migrating *VM4* to *Cn5* is impossible because this would again lead to a violation of the utilization threshold (67.5% + 25.7% > 90%), as would migrating *VM1* to any other compute node. Hence, three options remain: migrating *VM4* to either *Cn2* (`Depl_1`), *Cn3* (`Depl_2`) or *Cn4* (`Depl_3`). We can now use the simulation to predict the utilization of the VMs. The results show that in any case, the utilization values are below the threshold. This means that from the perspective of resource utilization, we have three possible adaptation options. But which one is better? Further constraints for the adaptation options results from the SLAs established with the different customers. Since the workload of the two customers did not change, system adaptation should have no implications on their SLAs. However, the predicted response times (see Table 5.7) show that migrating *VM4* has a significant impact on the response time of the customer whose VM has to share its resources. The results show that adaptation option `Depl_1` violates the SLA of the gold customer whereas SLAs are fine for `Depl_2`. `Depl_3` has no effect on the response times of the gold customers but the SLA of the silver customer `Cust8` is violated. Therefore, the only option to reconfigure the system without using additional resources is to migrate *VM4* to *Cn3* (`Depl_2`).

5.4.2. Quantifying Performance-Influencing Factors of Virtualization

We repeated the experiments on VMware ESX 4.0 to evaluate if the conclusions and the performance model derived from our analysis of Citrix XenServer 5.5 in Section 5.2.2.5 is applicable for other hypervisor architectures, too. VMware ESX is another industrial virtualization platform which, in contrast to Xen, is based on a monolithic hypervisor architecture, i.e., the hypervisor itself contains the device drivers and provides shared access to the physical devices. In the following sections, we compare the experiment results we obtained for the different virtualization platforms. We also provide a short experience report on the portability of our automated experimental analysis approach, which is of general interest when migrating from one virtualization platform to another as well as for automatic administration of virtualization platforms.

Overhead of Virtualization

After repeating the experiments on VMware ESX 4.0, we calculate the relative delta between the two platforms as $\frac{VMwareESX\ 4.0 - CitrixXenServer\ 5.5}{VMwareESX\ 4.0}$. The results in Table 5.8 show almost identical results for the CPU and memory benchmarks because both virtualization platforms use the hardware virtualization support. However, for the I/O benchmarks, VMware ESX 4.0 performs better. The reason for this is that in Citrix XenServer 5.5, all I/O workload is handled by the separate driver domain *Dom0*, which is less efficient than the monolithic architecture of VMware ESX 4.0. Hence, we can conclude that for CPU and memory, our performance model can be applied for VMware ESX 4.0, too. However, for the I/O performance overhead, it is important to distinguish these architectural differences.

Table 5.8.: Relative deviation of CPU, memory, disk I/O and network I/O results.

Benchmark	**rel. Delta**
CPU Mark	0.15%
Memory Mark	0.19%
Iperf, send	13.91%
Iperf, receive	15.94%
Disk Mark	19.14%

Scalability and Overcommitment

Concerning the performance behavior of VMware ESX 4.0 when scaling up and overcommitting, respectively, we observed a similar trend to the one on Citrix XenServer 5.5. Figure 5.25 shows this trend for the overcommitment scenario. As one can see, both platforms behave similarly, with slightly better scalability results for VMware ESX 4.0. Another observation was that on VMware ESX 4.0, using core affinity did not result in any performance improvements. This indicates an improved hypervisor scheduling strategy which takes care of multi-core environments and the cache and core effects we observed in Section 5.2.2.5.

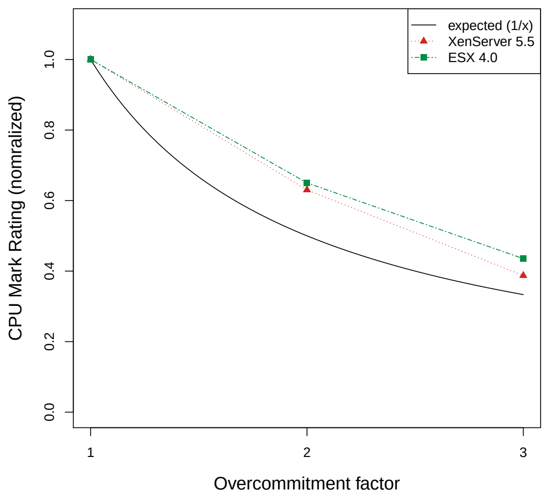


Figure 5.25.: Performance behavior in scenarios with CPU overcommitment for Citrix XenServer 5.5 and VMware ESX 4.0.

Mutual Influences of Workload Types

We also repeated the experiments to determine the mutual influences of workload types on VMware ESX 4.0. Table 5.9 lists the results. For CPU and memory intensive workloads, the observations are comparable to the ones for Citrix XenServer 5.5. Both workload types have an equal effect on each other caused by their similarities.

However, for disk intensive workloads, there is a big difference compared to XenServer 5.5. For VMware ESX 4.0, we observe a high performance degradation of the disk workload independent of the other workload type. For example, if the disk benchmark is executed with CPU or memory benchmark, disk benchmark results drop almost 50%, whereas CPU and memory benchmark results suffer from only 10% and 20% performance loss, respectively. One explanation is that, as VMware's virtual disk concept requires more CPU than the mechanism implemented in Xen, both VMs compete for CPU time. The different hypervisor architectures confirm this explanation. In the monolithic architecture of VMware ESX 4.0, the drivers are integrated in the hypervisor kernel, whereas they are moved to a separate domain in Xen.

Table 5.9.: Mutual performance degradation for different workload types on VMware ESX 4.0.

VM_A	CPU	CPU	Mem	CPU	Mem	Disk	CPU	Mem	Disk
VM_B	CPU	Mem	Mem	Disk	Disk	Disk	Net	Net	Net
r_A	47.03%	46.64%	49.23%	10.02%	17.21%	44.53%	9.95%	35.32%	14.87%
r_B	48.21%	40.29%	51.34%	49.56%	45.53%	44.82%	65.02%	54.56%	32.74%

Portability of the Automated Analysis

When porting our automated experimental analysis approach to VMware ESX 4.0, we gained some important experiences. These experiences are helpful when applying our approach for system adaptation, e.g., when migrating a VM from one platform to others.

The first challenge was that we could not install VMware ESX 4.0 on the HP Compaq machine because of the lack of driver support. The reason for lacking driver support is that supporting only commodity hardware keeps the monolithic hypervisor's footprint low. In Citrix XenServer 5.5's architecture, further drivers can be easily implemented in the Dom0 domain while still keeping the hypervisor's footprint small. Hence, depending on the requirements it might be worth selecting the hardware first and then the virtualization platform, or vice versa. The next technical challenge we faced was that the VMs of VMware ESX 4.0 are usually intended to be managed via external graphical tools, which hinders an automated approach. Fortunately, a special command line interface, which must be activated separately, can be used for automation. As both platforms support the *Open Virtualization Format (OVF)* for virtual machines, in theroy, porting the MasterVM should be easy. Although this standardized XML schema for OVF is in fact implemented on both platforms, they use a different XML tag semantic to describe the VM geometry. Hence, the export of a VM from Citrix XenServer 5.5 to VMware ESX 4.0 works in theory, but practically only with additional tools and workarounds, involving manual XML editing. Once migrated, one can reuse the concept of automated experimental analysis and experiment types, but one has to adapt the scripts to the target API. For example, the credit-based scheduler parameter *capacity* is named differently on VMware ESX 4.0.

In summary, we can conclude that the process we proposed is generic enough to be applied on other virtualization platforms, but some manual adjustments might be necessary. However, this is only a technical limitation given that currently there is no standardized virtual machine migration mechanism across different virtualization platforms.

5.5. Summary

In this chapter, we presented a meta-model to describe the resource landscape of modern dynamic IT infrastructures. The meta-model provides novel constructs to model the various resource layers of modern IT infrastructures as well as their performance-influencing factors. Additionally, we proposed a generic method for quantifying the performance-influencing factors of resource layers like virtualization. We applied this approach to two representative virtualization platforms, Citrix XenServer 5.5 and VMware ESX 4.0, to quantify the performance-influencing factors of virtualization and to evaluate the accuracy of our approach. We were able to confirm the results for CPU and memory intensive workloads as well as the observed trends in scalability and overcommitment scenarios. However, the experiments also showed that there are differences when handling I/O intensive workloads. Moreover, we presented a case study to illustrate the novel concepts of our resource landscape meta-model for run-time system adaptation. The results showed that modeling the resource landscape with its hierarchy provides valuable information important for run-time system adaptation and resource management, e.g., to exclude migration targets or to find the most suitable target.

6. Modeling System Adaptation Processes

In the software performance engineering community, a number of meta-models for building architecture-level performance models of software systems have been proposed over the last decade (Koziolek, 2010; Balsamo et al., 2004). Such models provide modeling constructs to capture the performance-relevant behavior of a system's software architecture and its execution environment. However, as they are typically designed for use at system design-time in an offline setting, they normally neglect the description of dynamic aspects of the system, such as varying deployments or adaptation processes, and are thus limited to analyzing steady states of the modeled system (Becker et al., 2012). As the effect of possible adaptation actions on the system performance and resource efficiency cannot be reflected, the use of such models as a basis for autonomic model-based system adaptation at run-time is insufficient. To fill this gap and provide a holistic model-based approach for engineering self-adaptive systems, the Descartes Modeling Language (DML) provides two further sub-meta-models presented in this chapter.

In the following, we introduce two novel meta-models, the adaptation points meta-model and the adaptation process meta-model. Together, these models form a generic and flexible formalism for modeling the dynamic aspects of self-adaptive systems as well as their adaptation processes. The adaptation points meta-model, described in Section 6.1, provides concepts to specify the degrees of freedom of architecture-level performance models. Concretely, we use the adaptation points meta-model to annotate architecture-level performance models and describe their valid configuration space, i.e., where and in what range such model instances can be adjusted. Based on the adaptation points meta-model, Section 6.2 presents a modeling language to describe adaptation processes of self-adaptive systems at the architecture-level in an intuitive and easily maintainable manner. Our meta-model distinguishes high-level adaptation objectives from low-level implementation details, explicitly separating system-specific adaptation operations from system-independent adaptation plans. In Section 6.3, we show the architecture of our adaptation framework which implements our concept of a model-based adaptation control loop, leveraging the distinct features of DML. Finally, Section 6.4 presents evaluation results, comparing our modeling approach with Story Diagrams (Fischer et al., 2000) and Stitch (Cheng and Garlan, 2012) and assessing its applicability.

6.1. Adaptation Points Meta-Model

Today's distributed IT systems are increasingly dynamic and offer various degrees of freedom for adapting the system at run-time. However, to realize model-based system adaptation, these properties must be reflected on the model-level. In this section, we introduce the adaptation points meta-model as part of the Descartes Modeling Language (DML). The aim of the adaptation points meta-model is to annotate architecture-level performance models to describe the degrees of freedom of the resource landscape and the application architecture, i.e., the points where the system can be adapted at run-time. In other words, adaptation points at the model level correspond to adaptation operations executable on the system at run-time. Other model elements that may change at run-time but cannot be influenced directly (e.g., the usage profile) are not in the focus of this meta-model. For example, changing the number of virtual CPUs assigned to VMs, migrating VMs or software components, or load-balancing requests, are adaptation points of an adaptive system that can be modeled with our adaptation points meta-model. In contrast, changes in the usage profile cannot be controlled. Thus, we do not consider them as adaptation points. From a high-level perspective, the adaptation points meta-model provides possibilities to specify the boundaries of the system's configuration space, i.e., it defines the possible valid states of the system architecture. However, it is not intended to specify how the actual change has to be performed on the model or the system. This is part of the adaptation process meta-model which builds on the adaptation points meta-model and will be introduced in Section 6.2.

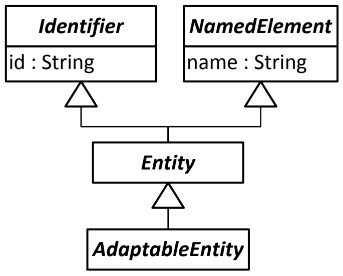


Figure 6.1.: Relation of Entity and AdaptableEntity.

The core question we face when designing an adaptation points meta-model is how to denote the entities in the resource landscape or application architecture meta-models that can be adapted at run-time. On the one hand, having an explicit type like AdaptableEntity (cf. Figure 6.1) in the meta-models makes it easier to specify adaptation points for a given model instance (e.g., a resource landscape) since the adaptable entities are already determined by their type (it is a sub-type of AdaptationEntity). Thus, the advantage is that it is not necessary to know all details of the resource landscape model instance to specify adaptation points. However, using AdaptableEntities as a type in the meta-model has the disadvantage that all adaptable entities must be specified already at meta-model design time, i.e., further entities cannot be added or removed without changing the meta-model. It is not always possible to distinguish adaptable entities from non-adaptable entities when designing the meta-model. For example, imagine the RuntimeEnvironment being of type AdaptableEntity. Then, any RuntimeEnvironment instance (hypervisors, VMs, JVMs) would be an adaptable entity, too. The problem is that adaptation points are usually system specific. For example, some systems support VM migration, whereas others do not. For these reasons, we also would like to be able to describe adaptation points for meta-model instances, not only at the meta-model level. This is a difference compared to the approach of Koziolek and Reussner (2011) that focuses on the meta-model level to describe which variants of model instances can be created at design time.

The question of how to denote adaptable entities and on which abstraction level of the MOF standard (Object Management Group (OMG), 2011b) is also related to the question where to introduce them, i.e., in which meta-model(s). Since the application architecture can be adapted as well, it is insufficient to introduce adaptation point modeling constructs only in the resource landscape meta-model. Hence, to avoid having such constructs in both meta-models, we decided to introduce our adaptation points concepts in a separate meta-model. This has the advantage that aspects related to the static system architecture (resource landscape and application architecture) are decoupled from aspects related to system adaptation. This separates knowledge about *what* can be adapted from *how* to execute the adaptation. Furthermore, using a separate adaptation points meta-model also has the advantage that resource allocation algorithms or system adaptation processes can refer to adaptation points instead of operating on the model instance directly. Thereby, information about where to change the model instance is not disclosed directly to the person or program adapting the model. Instead, such information is considered as an explicit entity of the model-based system adaptation process. Furthermore, the explicit definition of adaptation points helps to specify valid system configurations. By using specified adaptation points within system adaptation processes, inconsistent system states can be avoided. Another benefit of this separation is the support of reuse and improved maintainability. For example, the adaptation point descriptions in our example might be reused for other model instances, or for different adaptation processes.

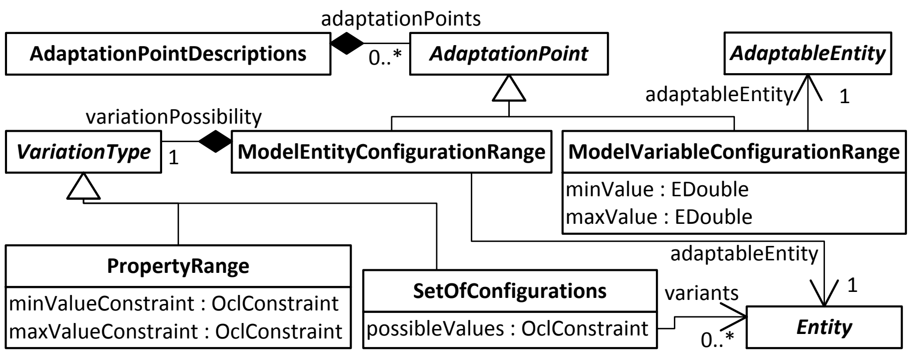


Figure 6.2.: Adaptation points meta-model.

Figure 6.2 depicts the structure of the adaptation points meta-model. The root element AdaptationPointDescriptions collects all adaptation point annotations for adaptable elements of architecture-level performance models. We distinguish between two types of adaptations we can perform on the architecture-level performance model: a) changing attribute values of model entities, e.g., the value of a PassiveResourceCapacity and b) changing the number of instances of a model entity, e.g., a RuntimeEnvironment. These two types of AdaptationPoints are modeled as ModelVariableConfigurationRange and ModelEntityConfigurationRange, respectively. Note that in our approach, we consider only those parts of a model to be adaptable that are actually annotated by an AdaptationPoint. The reason is that even if a system technically supports certain ways to be adapted, there might exist an instance of the system where it is prohibited to change its configuration. An illustrating example is given by virtualized systems where in general, the number of virtual resources assigned to virtual machines can be changed at run-time. However, in some systems, this feature might be deactivated for reliability reasons.

A ModelVariableConfigurationRange refers to an AdaptableEntity and specifies the range in which the attribute value of this AdaptableEntity can be altered, restricted by minValue and maxValue attributes. An AdaptableEntity is a specialization of the abstract class Entity (cf. Figure 6.1). The type Entity is just a convenience class and almost all classes of DML are sub-classes of this abstract class to inherit the name and id attributes. The difference of AdaptableEntity compared to Entity is the following. If a meta-model class extends AdaptableEntity, this denotes that the attribute values of this particular meta-model class are explicitly adaptable. It is the responsibility of the AdaptableEntity to indicate which attribute of its child is adaptable. All types with an attribute that is explicitly adaptable at run-time are modeled as a sub-type of AdaptableEntity. An example for such an AdaptableEntity is the NumberOfParallelProcessingUnits of a configuration specification (cf. Figure 5.3).

The ModelEntityConfigurationRange can be used to annotate other architecture-level performance model instance entities that are not a sub-type of AdaptableEntity, e.g., the instances of a specific RuntimeEnvironment. The ModelEntityConfigurationRange refers to an Entity, denoting that this referred Entity can be adapted. An Entity can be any entity of an architecture-level performance model instance, e.g., a Container. The VariationType of the ModelEntityConfigurationRange specifies in more detail how this model entity can vary. Currently, we distinguish two variation types: PropertyRange and SetOfConfigurations. In contrast to the ModelVariableConfigurationRange where we specify an attribute value range, the idea of the PropertyRange is to specify a range for the referred Entity. We use Object Constraint Language (OCL) constraints (minValueConstraint and maxValueConstraint) to check whether the variation is within the valid value range or not. For example, think of a constraint to set a minimum and maximum amount of VM instances on a server. The SetOfConfigurations can be used to model any other kind of variability that has no order or range, e.g., the deployment of a VM (Container) on a set of target hosts (also Containers). In this case, possible target model instances are collected in the SetOfConfigurations, which is a list of other Entities. In the example of the VM deployment, this set can contain the references to different Container instances, i.e., a list of target hosts where the VM can be deployed on.

In summary, the adaptation points meta-model describes the degrees of freedom and the configuration space of modern IT systems, specifying adaptation points in architecture-level performance models. The AdaptableEntity can be used to denote adaptable entities on the meta-model level, whereas ModelEntityConfigurationRange provides the means for denoting an adaptable entity on the model instance level. These concepts are not intended to describe all possible instance variants a system might have but to specify a boundary within which system adaptation processes can operate. How to model these adaptation processes based on the adaptation points meta-model will be explained in Section 6.2.

Example

The following example illustrates how to use the adaptation points meta-model to specify the adaptable parts of the example resource landscape model instance introduced in Section 5.1.5. Figure 6.3 depicts the example model instance. The variable resources and model entities we consider here are the number of virtual CPUs (vCPUs) of a VM (*NrOfVcpus*), the number of VM instances (*VmInstances*), and the location of a VM (*VmHost*).

Corresponding to these variable elements, the adaptation points model instance contains three different adaptation points, one ModelVariableConfigurationRange and two ModelEntityConfigurationRange. *NrOfVcpus* is of type ModelVariableConfigurationRange and specifies a configuration range in which the number of virtual CPUs of *VmTemplate* can be varied, limited by minValue = 2 and maxValue = 4. Also important to note is that the adaptation point refers to a ContainerTemplate. This way, we can express that all vCPUs of all VMs referring to this template can be varied in this range. Referring directly to a Container means that only the attribute value of this specific container is adaptable.

The second adaptation point we annotate in our example resource landscape model instance is called *VmInstances* and is an example for the variation type PropertyRange. It

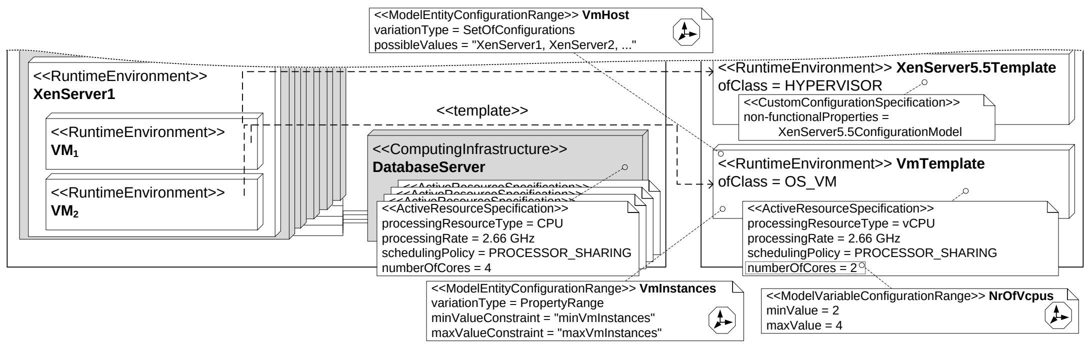

<<ModelEntityConfigurationRange>> VmHost
variationType = SetOfConfigurations
possibleValues = "XenServer1, XenServer2, ..."
<<RuntimeEnvironment>>
XenServer1
<<RuntimeEnvironment>>
VM₁
<<RuntimeEnvironment>>
VM₂
<<template>>
<<ComputingInfrastructure>>
DatabaseServer
<<ActiveResourceSpecification>>
processingResourceType = CPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 4
<<RuntimeEnvironment>> XenServer5.5Template
ofClass = HYPERVISOR
<<CustomConfigurationSpecification>>
non-functionalProperties =
XenServer5.5ConfigurationModel
<<RuntimeEnvironment>> VmTemplate
ofClass = OS_VM
<<ActiveResourceSpecification>>
processingResourceType = vCPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 2
<<ModelEntityConfigurationRange>> VmInstances
variationType = PropertyRange
minValueConstraint = "minVmInstances"
maxValueConstraint = "maxVmInstances"
<<ModelVariableConfigurationRange>> NrOfVcpus
minValue = 2
maxValue = 4

Figure 6.3.: Adaptation points meta-model instance annotating the resource landscape model example in Figure 5.6.

refers to the RuntimeEnvironment template *VmTemplate* and specifies a PropertyRange using OCL constraints (cf. Listing 6.1).

```
context ModelEntityConfigurationRange
inv minVmInstances:
    let similarContainers : Set(Container) = Container.allInstances()
           -> select(c | c.template = self.adaptableEntity)
    in similarContainers -> size() > 1;

context ModelEntityConfigurationRange
inv maxVmInstances:
    let similarContainers : Set(Container) = Container.allInstances()
           -> select(c | c.template = self.adaptableEntity)
    in similarContainers -> size() < 4;
```

Listing 6.1: OCL constraints of *VmInstances*.

Both OCL constraints query for all Container instances and select those which refer to the same template as the ModelEntityConfigurationRange. The resulting number must be above one and below four, respectively. The OCL constraints ensure that each *XenServer* contains at least one VM and not more than four. Note that for the correct evaluation of the OCL constraints the context of the OCL constraint must be set to the ModelEntityConfigurationRange instance which actually refers to the model instance entity to be evaluated. This is important because the context also limits the scope of the adaptation possibilities.

The third adaptation point *VmHost* is an example of the variation type SetOfConfigurations. Its purpose is to specify a changeable location of a VM which can be used for modeling a VM migration. Therefore, it refers to the RuntimeEnvironment template *VmTemplate* as the adaptable model entity. Furthermore, its attribute list possibleValues refers to a set of target hosts (*XenServer1*, . . . , *XenServerN*). This list denotes the possible target hosts for the referred entity *VmTemplate*.

6.2. Adaptation Process Meta-Model

Any self-adaptive system follows a certain kind of process with the purpose to adapt itself to changes in its environment such that operational goals are continuously fulfilled. In this context, operational goals are either system-wide defined QoS properties the system has to fulfill or user-specific SLAs (cf. Chapter 2). The adaptation process meta-model we present in the following is a modeling language to specify such adaptation processes. In our meta-model, we define three main elements—*Strategy*, *Tactic*, and *Action*—to describe the adaptation process at three different levels of abstraction (cf. Figure 6.4). We also refer to this meta-model as Strategies/Tactics/Actions (S/T/A) adaptation language. At the highest abstraction level are the strategies. Strategies aim at achieving a given high-level objective by applying one or more tactics that are defined for this strategy. Tactics are more system-specific and pursue a short-term goal by executing one or more adaptation actions. Actions implement the actual adaptation operations on the system model or on the real system, respectively. Thus, they are the technical part of the adaptation process, encapsulating system-specific details.

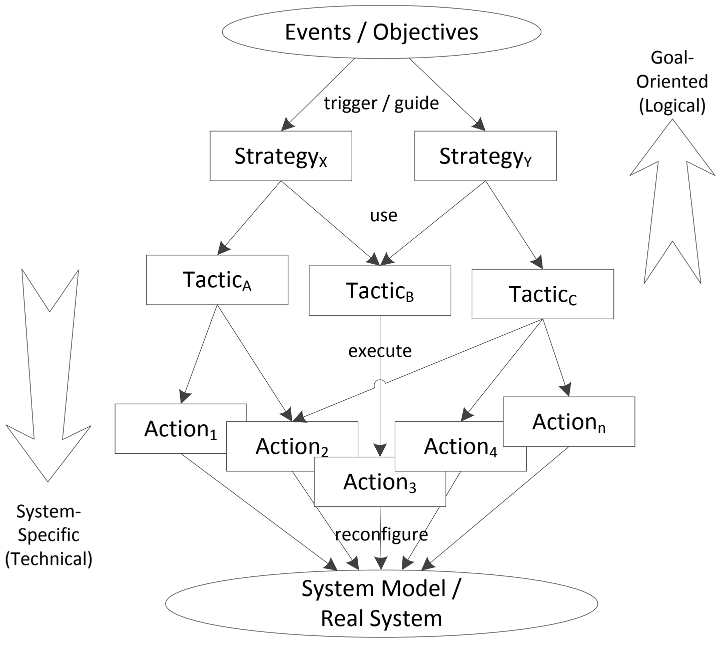


Figure 6.4.: Main concepts of the adaptation process meta-model and their relations, bridging different abstraction levels.

The novelty and important advantage of this modeling approach is that it distinguishes high-level goal-oriented objectives (strategies) from low-level system-specific details (adaptation tactics and actions) and explicitly separates platform specific adaptation operations from system-independent adaptation plans.

Before we explain the modeling abstractions of S/T/A in detail, we want to emphasize the conceptual difference between strategies, tactics, and actions. A strategy captures the *logical* goal-oriented aspect of an adaptation process. It defines the objective that needs to be accomplished and describes the possible ways to achieve this objective. A strategy can be a complex, multi-layered plan for accomplishing the objective. However, which step is taken next will depend on the current state of the system. Thus, in the beginning, the sequence of tactics applied by the strategy is unknown. Which tactic is applied next depends on the impact of the tactic, which is evaluated using the online performance prediction capabilities of DML. This gives us the flexibility to react in unforeseen situations and switch to different tactics. To give some examples, a defensive strategy for resolving a resource bottleneck could be "*add as few resources as possible stepwise until response time violations are resolved*," whereas an aggressive strategy would be "*add a large amount of resources in one step so that response time violations are eliminated, ignoring resource efficiency.*"

In contrast to strategies, tactics and actions realize the *technical* aspect that follows the planning of an adaptation. While strategies are focused on how to act, i.e., on deciding which tactic might be most effective w.r.t. the strategy's objective given the current system state, tactics specify precisely which actions to take without explicitly considering the effect. Therefore, tactics define a specific sequence of actions and initiate the execution of these actions. Furthermore, we define tactics with the following intrinsic semantics, inspired by the semantics of transactions in database systems: i) atomicity, i.e., either the whole tactic with all its contained actions is executed or the tactic must be rolled back, ii) consistency, i.e., the model's state must be consistent after applying a tactic, and iii) determinism, i.e., tactics have the same output when applied on the same model state. The motivation for the above definition of tactics is to group and execute multiple actions in an atomic manner leaving the system model in a consistent state. This is important because after applying a tactic, the effect of a tactic is evaluated leveraging DML's online performance prediction capabilities to analyze the tactic's impact. This impact influences how the strategy continues to adapt the system. After applying the tactic on the model, we evaluate the impact of the tactic using online performance prediction techniques. If the application of a tactic is predicted to contribute towards achieving the pursued adaptation goal, the tactic is maintained as part of the constructed concrete adaptation plan to be executed on the real system later. Otherwise, the tactic is rolled back and another tactic is applied. The configuration of the real system is only changed once we have found a model state that is predicted to satisfy the adaptation goal. How we actually implement this behavior is explained in Section 6.3 as part of the realization of the adaptation framework.

The idea of distinguishing three abstraction levels is a valid concept and can be found in other approaches, too (cf. Section 3.1.2). However, these approaches either do not consider an end-to-end model-based approach or have limited expressiveness. In contrast to existing approaches, we propose a generic meta-model explicitly defining the relation of strategies, tactics and actions to describe adaptation processes at the architecture-level in an intuitive and easily maintainable manner while still providing the flexibility to react in situations of uncertainty. In the following, we describe the concepts of our adaptation process meta-model bottom-up, beginning with the actions (cf. Figure 6.4).

6.2.1. Actions

In Figure 6.5, we depict the meta-model of our adaptation language. Actions are the atomic elements on the lowest level of the adaptation process' hierarchy (cf. Figure 6.4). They represent the execution of an adaptation operation on the model or the real system, respectively. Actions can refer to Parameters to specify a set of input and output parameters. A parameter is specified by its name and type. Parameters can be used to customize the action, e.g., to specify the source and target of a migration action or to use return values (output parameters) of executed actions as input parameters for subsequent actions.

To model an adaptation operation, actions refer to adaptation points that have been specified with the adaptation points meta-model. However, they do not specify how the operation is actually implemented, neither at the model nor at the system level. The interpretation of the modeled action and the implementation of the actual adaptation operation is the responsibility of the adaptation framework interpreting the adaptation process model instance. This is important to separate technical system-specific details from logical aspects. In Section 6.3, we will present an adaptation framework that realizes the interpretations of adaptation actions for a given resource landscape model instance. To provide further semantics about how to interpret and perform the adaptation operation, an Action contains an AdaptationActionOperation. The AdaptationActionOperation describes the direction and scope of the adaptation operation. The AdaptationOperationDirection specifies in which "direction" to execute the adaptation. Currently, we support

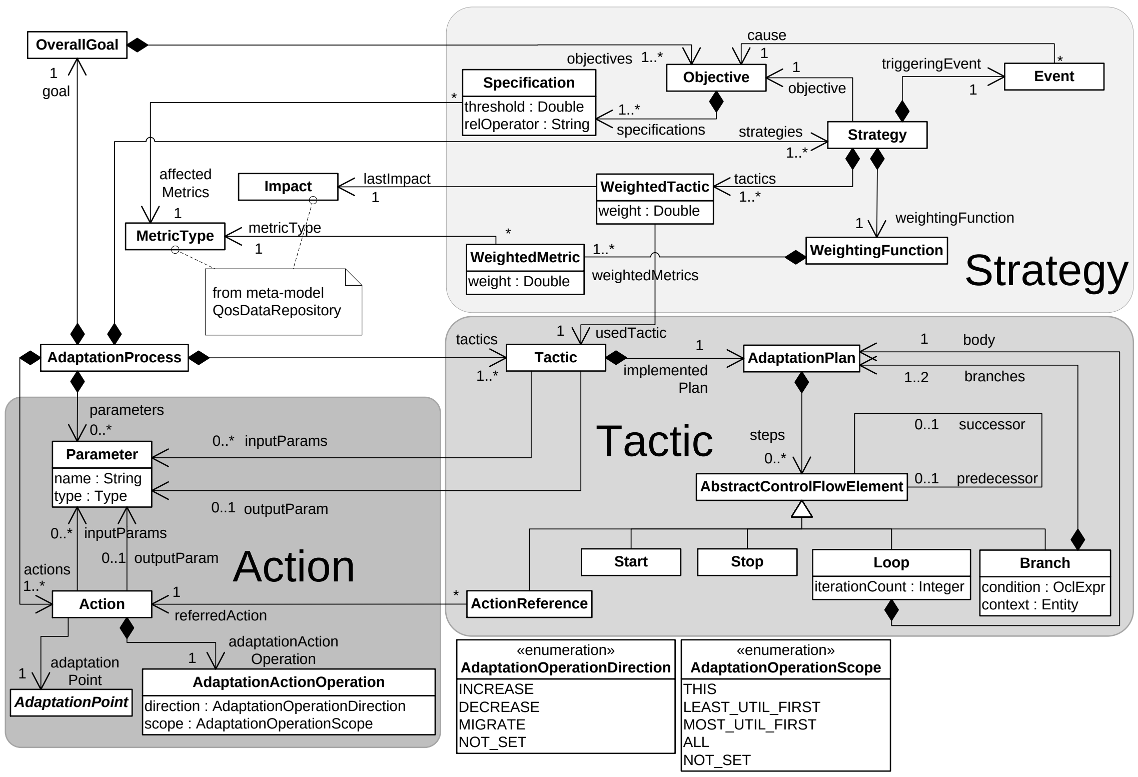


Figure 6.5.: Adaptation process meta-model.

four different modes: INCREASE, DECREASE, MIGRATE, and NOT_SET. For example, INCREASE indicates to increase the attribute value of an AdaptableEntity or to scale up the number of instances of a model entity, whereas MIGRATE indicates to move an Entity. The AdaptationOperationScope specifies where to apply the adaptation operation. This is important in case the adaptation operation can be applied to multiple model entities. For example, if the adaptation point refers to a ContainerTemplate, the scope indicates if the adaptation operation has to be applied to, e.g., ALL instances or the least utilized LEAST_UTIL_FIRST instance of the set of Containers referring to this ContainerTemplate. The mode THIS can be used to indicate that exactly this entity has to be changed. The list of modes is extensible, however, one must also extend the adaptation framework to support newly introduced modes.

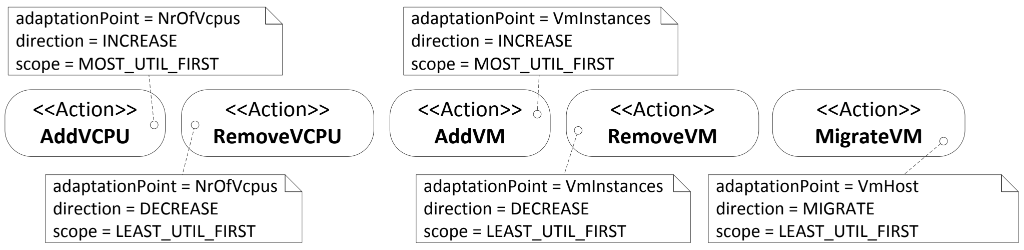


Figure 6.6.: Example Actions referring to adaptation points.

Example: Figure 6.6 shows five example actions. The type of actions that can be modeled depends on the types of adaptation points that have been defined for the respective architecture-level performance model, depicted in Figure 6.3. The actions *AddVCPU* and *AddVM* can be used to increase the amount of allocated resources in a system, either by adding vCPUs to a VM (*AddVCPU*) or by adding new VMs (*AddVM*). Similarly, *Re-*

moveVCPU and *RemoveVM* can be used to remove resources. *MigrateVM* can be used to move VMs between hosts. To provide the adaptation framework with more details about how to execute the respective modeled action, these actions also specify an AdaptationOperationDirection and AdaptationOperationScope. In our example, the directions are either INCREASE or DECREASE, indicating to increase or decrease the vCPUs parameter or the number of VM instances. The scopes indicate that either the least or most utilized container instance referring to the container template *VmTemplate* are the candidate to execute the adaptation operation. For the action MigrateVM, we specified the AdaptationOperationDirection *MIGRATE* to indicate that the adaptation framework should move the model entity the adaptation point *VmHost* refers to. The scope *LEAST_UTIL_FIRST* specifies that the target host for the migration is the least utilized host from the SetOfConfigurations specified by the adaptation point.

6.2.2. Tactics

When modeling adaptation processes, each Tactic has a certain purpose (e.g., to scale-up resources) expressed by its specific AdaptationPlan. The AdaptationPlan describes a process of how the tactic pursues its purpose, i.e., in which order to apply actions to adapt the system. Therefore, each AdaptationPlan contains a set of AbstractControlFlowElements. The order of these control flow elements is determined by their predecessor and successor relations. Concrete types of the AbstractControlFlowElement are Start and Stop as well as Loop and Branch. They describe the control flow of the adaptation plan. Start and Stop denote the beginning and end of the adaptation plan. The Loop element can be used to specify that the adaptation plan in the body of the loop will be executed n times, whereas n is given by the attribute iterationCount. A Branch has the semantic of conditional statement. Its intention is to influence the control flow of the adaptation plan depending on the current model state. Therefore, it has two attributes, condition and context. In this thesis, a condition is an OCL expression (invariants) which evaluates to true or false depending on the current state of the model. The context for evaluating the OCL invariant is given by the attribute context of the Branch. Furthermore, tactics can refer to Parameters to specify input or output parameters. These parameters can be evaluated to influence the control flow, e.g., by specifying iteration counts. Actions are integrated into the control flow by the ActionReference entity.

The advantage of the tactic's AdaptationPlan concept is that AdaptationPlans specify a complex but deterministic part of the adaptation process. This ensures that the previously mentioned requirement, that Tactics are *deterministic*, is fulfilled. Furthermore, the execution of an AdaptationPlan is only complete if all of its sub-steps have been completed. If any adaptation action on the model fails, the model can be reset to the state before starting the AdaptationPlan. This ensures the *atomicity* property. After executing an AdaptationPlan, we can also check if the adapted model is valid to ensure the *consistency* property of Tactics. Given that our model-based system adaptation process relies on model analysis to guide the adaptation process, it is important to regularly check the impact of adaptation actions on the system performance. However, model analysis can be costly, which is why we decided to conduct model analysis only after applying a tactic. Thus, the AdaptationPlan is also a way to bundle several model adaptation actions to save costly analysis steps.

Example: In Figure 6.7, we show three example tactics using the previously presented actions *AddResources*, *RemoveResources*, and *MigrateVM*. The purpose of these tactics is to increase system resources, e.g., to maintain SLAs, or to consolidate the system resources to increase efficiency.

The adaptation plan of the tactic *AddResources* implements a Loop action executed as many times as specified in iterations, which is an input parameter to this tactic. With this

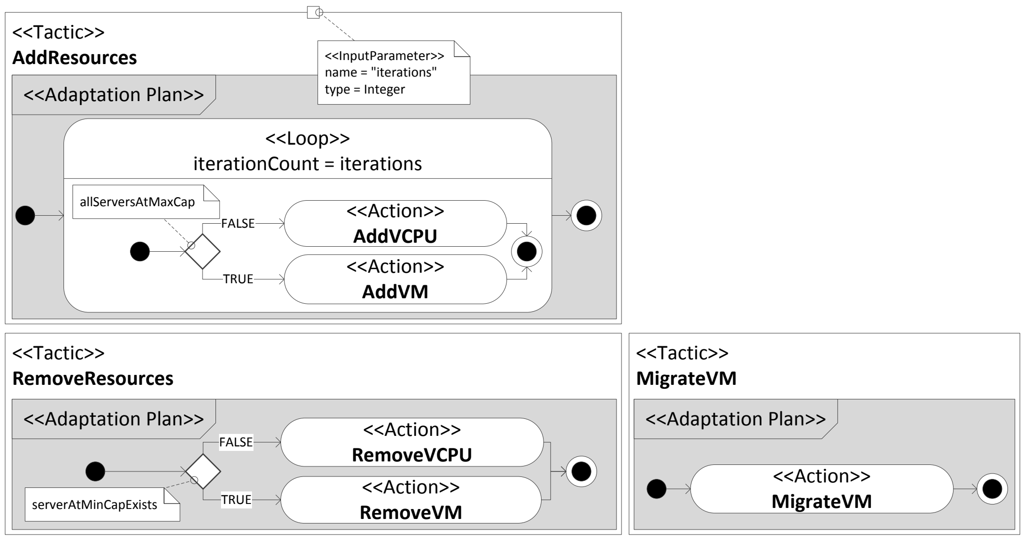


Figure 6.7.: Different example Tactics using the previously specified Actions.

parameter one can specify how many resources to add by executing the tactic. The body of the Loop action implements two actions, *AddVCPU* and *AddVM*. Which action is executed depends on the current state of the underlying architecture-level performance model. In this example, a Branch with the condition `allServersAtMaxCap` ensures that the NumberOfParallelProcessingUnits is below the maximum value of four. The OCL constraint for the condition `allServersAtMaxCap` is given in Listing 6.2. If this constraint evaluates to true, the *AddVCPU* action is executed to add an additional vCPU to a VM. Else, the *AddVM* action adds a new VM. The adaptation plan of this tactic is also a good example to illustrate the separation of technical and logical details because the tactic specifies that resources should be added but it does not specify how to implement this. This is covered by the framework executing the tactic (cf. Section 6.3).

```
context RuntimeEnvironment
inv allServersAtMaxCap:
        RuntimeEnvironment.allInstances()
         -> select(re | re.template = self.template)
            -> exists(re | re.configSpec.oclAsType(
                resourceconfiguration::ActiveResourceSpecification)
                .processingResourceSpecifications
                   -> forAll(nrOfParProcUnits.number < 4)
        and
        RuntimeEnvironment.allInstances()
         ->  select(re | re.template = self.template)
             -> forAll(re | re.template.templateConfig.oclAsType(
                 resourceconfiguration::ActiveResourceSpecification)
                 .processingResourceSpecifications
                    -> forAll(nrOfParProcUnits.number < 4)
```

Listing 6.2: OCL invariant `allServersAtMaxCap` of the AddResources tactic.

The adaptation plan of the tactic *RemoveResources* either removes a VM if there is a runtime environment running at minimum capacity or removes a vCPUs from a VM, otherwise. The VM to be removed is determined by the OCL constraint identifying the VM running at minimum capacity (Listing 6.3). This tactic can be considered as a conservative tactic, as it removes no more than one resource unit at a time. To remove further resources, the tactic must be executed again. The advantage of this conservative tactic is that it reduces the resources stepwise and after each step (i.e., after applying one tactic), the impact of the tactic is evaluated.

```
context RuntimeEnvironment
inv serverAtMinCapExists:
        RuntimeEnvironment.allInstances()
         -> select(re | re.template = self.template
                        and not re.configSpec -> isEmpty())
            ->exists(re | re.configSpec.oclAsType(
                resourceconfiguration::ActiveResourceSpecification)
                .processingResourceSpecifications
                   -> forAll(nrOfParProcUnits.number > 1))
```

Listing 6.3: OCL invariant `serverAtMinCapExists` of the RemoveResources tactic.

Tactic *MigrateVM* contains an adaptation plan with only one action, *MigrateVM*. This tactic's purpose is to increase resource efficiency by migrating VMs.

6.2.3. Strategies

Any modeled adaptation process pursues an overall goal consisting of one or more different Objectives. The purpose of a Strategy is to achieve an Objective. An objective contains one or more Specifications to express the objective in a machine processable way (e.g., avg. response time of service x $< \tau$). A Specification refers to a MetricType and defines a threshold τ for this metric type. The specification also contains a relational operator relOperator (like $>, \leq, =$) that determines how to compare the metric type with the threshold. The specifications will be used later when evaluating the impact of the tactics used by the strategy. Note that other, more complex Specifications like goal policies or utility functions referring to multiple metric types (e.g., resource usage vs. utilization) can be added here, too. All objectives are collected within the OverallGoal. The OverallGoal has no explicitly defined semantics. It serves as a human-readable description of the overall goal of the adaptation process. Note that it is explicitly allowed to have multiple alternative strategies for the same objective because strategies might differ in their implementation but pursue the same objective.

The execution of a strategy is triggered by a specific Event that occurs during system operation, e.g., an event emitted periodically to maintain system resource efficiency or an event caused when a given Objective is violated. Such events trigger the execution of the strategy they are associated with to ensure that the objective of the strategy is achieved. In our approach, we assume that events occur sequentially and that they trigger only one strategy. This avoids inconsistencies through multiple strategies operating at the same time with possibly conflicting objectives. However, this does not limit our approach to scenarios without conflicting objectives. In our approach, we can handle such situations by designing strategies that express the conflicting objectives as utility functions like in the example by Kephart and Walsh (2004). The respective strategy in such a situation must contain suitable tactics and apply them such that the trade-off expressed by the utility function is achieved.

To achieve its objective, a strategy can choose from a set of WeightedTactics. Which tactic it uses depends on the current weight of the tactics, which is determined by the impact the tactic achieved when executed. These weights are calculated according to the strategy's WeightingFunction, which is explained in Section 6.2.5. The reason why we use weighted tactics is that this concept introduces a certain amount of indeterminism at a higher abstraction level. Having this indeterminism at the strategy level provides flexibility to find new solutions if a tactic turns out to be inappropriate for the current system state.

Example: Figure 6.8 depicts two example strategies. The first strategy is the *ResolveBottleneck* strategy with the objective to improve response times to maintain SLAs (90% quantile of response time $rt_x < 500$ ms), and a *ReduceResources* strategy with the objective to optimize resource efficiency (OverallUtilization > 60%). To specify these objectives on the model level, their specification refer to the respective MetricTypes. The *ResolveBottleneck* strategy uses only one tactic, namely *AddResources*, and is triggered by the *SlaViolated* event. After the tactic has been successfully applied at the model level, the architecture-level performance model is analyzed to predict the impact on the metric type referred by the objective. If the prediction results still reveal SLA violations, the strategy executes the tactic again until all SLA violations are resolved and the strategy has reached its objective.

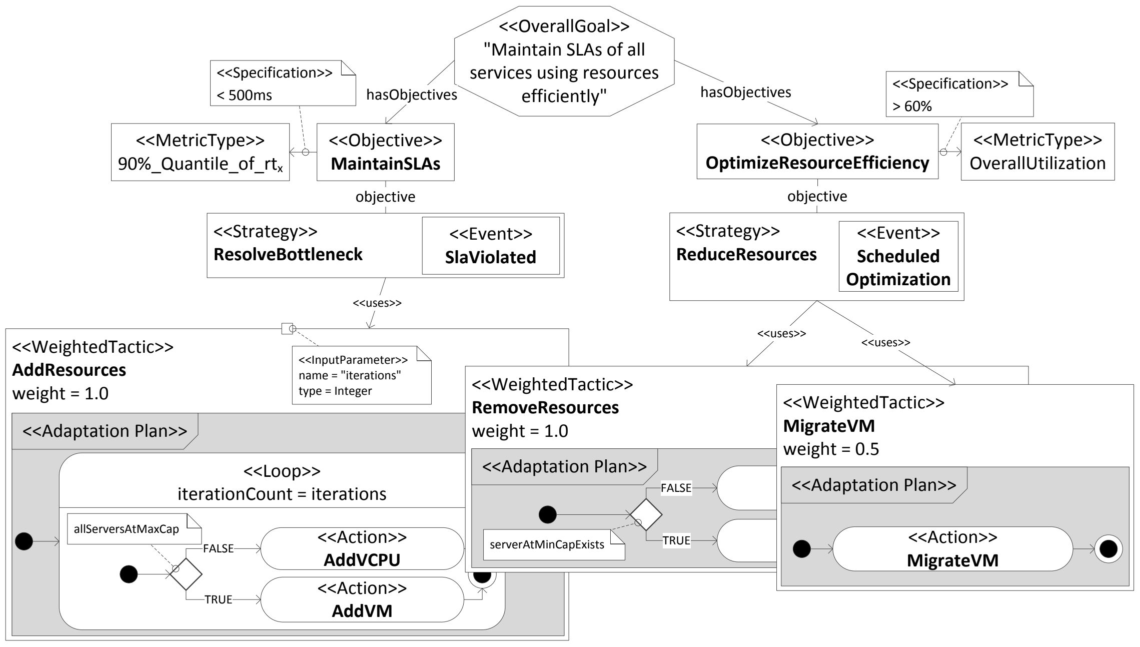


Figure 6.8.: Example Strategies using Tactics with assigned weights.

The *ReduceResources* strategy is triggered with the objective *OptimizeResourceEfficiency*. The trigger of the related event could be, e.g., a predefined schedule. The *ReduceResources* strategy refers to two tactics. *RemoveResources* reduces the amount of resources used by the system whereas *MigrateVM* aims at increasing resource efficiency by consolidating VMs. After the execution of the tactic, the underlying architecture-level performance model is analyzed to predict the effect of the tactic on the system performance. If no SLA violation is detected, the strategy can continue removing or consolidating resources. In case an SLA violation occurs, the application of the last tactic must be reverted, i.e., the adaptation framework must undo the adaptation actions of the tactic's adaptation plan. Which of these two tactics is chosen depends on their current weights. In our example, the initial values are 1.0 for *RemoveResources* and 0.5 for *MigrateVM*. The concepts of the weighting function will be presented with more details in Section 6.2.5.

6.2.4. QoS Data Repository

To evaluate the effect of executed tactics, we use QoS-related metrics that can be measured at the model and system level, respectively. We collect and store such measurements in a repository that can be queried later, e.g., to quantify the effect of tactics on metrics that are relevant for the strategy's objective.

This repository is called QoSDataRepository (cf. Figure 6.9). It contains a set of MetricTypes that can be obtained from the model or system, respectively. The MetricTypes are identified by their name. Examples for such MetricTypes are the average response time of $service_x$, the 90% quantile of response time of $service_y$, or the average utilization of $resource_n$.

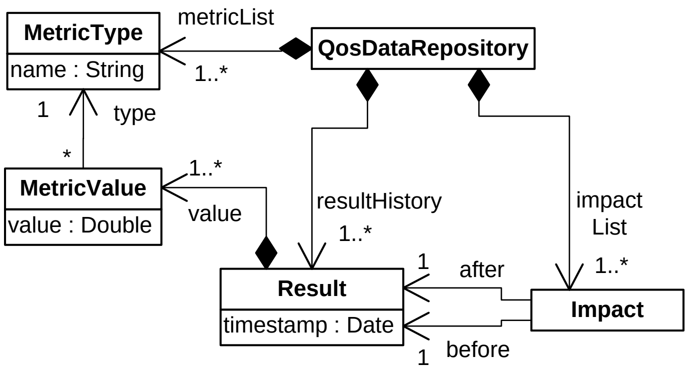


Figure 6.9.: The QoS data repository meta-model.

The repository also contains a history of Results. A Result is a set of MetricValues collected at a given point in time (timestamp). A MetricValue contains the actual value of a MetricType at this time point. Based on this information we can describe the achieved Impact of a tactic as the difference between two Results that have been obtained before and after the application of a tactic. For example, if the value of a metric was 500 ms and is 200 ms after the execution, the impact would be -300 ms, i.e., an improvement of the response time metric.

Our intention was not to define a new meta-model for QoS metrics and values but rather a quick and easy way to query QoS-relevant data. Thus, we kept this meta-model very basic to adapt and re-use it in other scenarios. For example, this meta-model can serve as a decorator model for the Structured Metrics Metamodel (SMM), developed by the Architecture-Driven Modernization Task Force of the Object Management Group (OMG) (2012). Thereby, it is easier to reuse other existing tools based on SMM, e.g., the MAMBA Execution Engine and Query Language of Frey et al. (2012). When decorating SMM, the class MetricType refers to the ObservedMeasure of the SMM, a MetricValue corresponds to the Measurement of the SMM, and Result corresponds to Observation. In this way, we can use data that has already been collected and stored in a repository by measurement tools like Kieker (van Hoorn et al., 2012). Additionally, for this meta-model we can easily provide a connector for the Query Engine developed by Gorsler et al. (2014); Gorsler (2013). Then, one can use the Descartes Query Language (DQL) (Gorsler et al., 2014) to formulate SQL-like statements to easily query relevant performance data from our repository.

6.2.5. Weighting Function

To decide which tactic to apply next, a strategy chooses the tactic that has the highest weight. The weight is calculated and assigned directly after executing the tactic, so the strategy might choose a different tactic in the next adaptation step. The actual value of the weight of a tactic depends on the impact the tactic achieved in the adaptation process, i.e., if metrics of interest have been improved or degraded. How to calculate such weights can be specified with a WeightingFunction. In our context, a weighting function is formally specified as follows. Let

$T = \{t_1, t_2, \ldots, t_l\}$ be the set of tactics,

$M = \{m_1, m_2, \ldots, m_m\}$ be the set of metric types,

$S = \{1, 2, \ldots, n\}$ denote the adaptation iterations of the system adaptation process, and

$V_s = \{v_s(m_1), v_s(m_2), \ldots, v_s(m_m)\}$ be the set of metric values at adaptation step $s \in S$.

Then, we define a weighting function as

$$f : T \times S \to \mathbb{R},$$

i.e., function that assigns a real number (the weight) to the given tactic t at a given adaptation step s. The idea is that any existing optimization algorithms or meta-heuristics like Tabu Search or Simulated Annealing (cf. Blum and Roli, 2003) can be used here to determine the weights depending on the current state of the system. As future work, it is also possible do define weighting functions that also consider a certain number of previous system states.

Currently, we specify weighting functions that are based on performance metrics. However, our approach can be easily extended with other more complex weighting functions and weighting functions for other QoS properties. Our WeightingFunction uses one or more WeightedMetrics (cf. Figure 6.5) to calculate the weight for a tactic. A WeightedMetric assigns a specific weight to the referred MetricType. More formally, we define a weighting function $w : M \to \mathbb{R}$ that assigns a real number to any metric type $m \in M$. The weight of a metric type can be used to express the importance of this metric type to the overall result of the weighting function. In contrast to the weights of tactics, this weight assigned to metrics is specified when designing the adaptation process and is fixed during the adaptation process. In a weighting function, the weights for metrics are used as follows. To determine the weight for an applied tactic t, we calculate the achieved Impact of t on each metric $m \in M_t$ and multiply it with the weight that is assigned to m, where $M_t \subseteq M$, containing only the metric types affected by t. More formally, the weight for a tactic t is calculated as

$$f(t, s) = \sum_{m \in M_t} i_s(m) \cdot w(m)$$

where $i_s(m)$ is the impact of tactic t on metric type $m \in M_t$ and $w(m)$ is the weight assigned to metric type $m \in M_t$. To quantify this impact, we calculate the delta of the values of the relevant metric types at the timestamps before and after the application of a tactic

$$i_s(m) = v_s(m) - v_{s-1}(m).$$

In other words, the weights of the set of metric types multiplied with the impact of the applied tactic on these metric types determines the weight that is assigned to the tactic. The reason why we use weights is that they are machine-processable and they relate directly to our specification of Objectives, which also refer to the same MetricTypes. The specification of the WeightingFunction can be based on or even derived from the strategy's objective. However, this (automatic) derivation is part of future work.

Example: To illustrate our WeightingFunction concept, we give two example weighting functions for our example strategies depicted in Figure 6.8. Imagine that the *ResolveBottleneck* strategy's objective is to maintain SLAs, but it should prioritize tactics that have a beneficial impact on the services of the more important customers. Therefore, assume that we have two different services, one of a gold customer ($service_{gold}$) and one of a silver customer ($service_{silver}$). For each of these services, we observe the metric types *90% quantile of the response time*, i.e., $M = \{rt_{gold}, rt_{silver}\}$. To prioritize the impact on the response time of the gold customer's service over the one of the silver customer, we set the weighted metrics to $w(rt_{gold}) = -2.0$ and $w(rt_{silver}) = -1.0$. Note that the weights are negative because improving the response time results in a negative impact. To assure that a tactic which is beneficial for the gold service gets a higher weight, we can specify a weighting function

$$\forall s \in S : f_{ResolveBottleneck}(t, s) := \sum_{m \in M} w(m) i_s(m)$$

that assigns weights depending on the impact on the response times of the gold and silver customer, respectively.

Another example is the weighting function we specified for the *ReduceResources* strategy to assign new weights to the tactic *RemoveResources* and *MigrateVM*.

$$\forall s \in S : f_{ReduceResources}(t, s) := \begin{cases} 1, & \text{if } rt_{gold} < \tau \wedge rt_{silver} < \tau \\ 0, & \text{else.} \end{cases}$$

This function assigns a weight of one to the given tactic t as long as the response time metrics M are below the given threshold τ, i.e., the SLAs are not violated. Only if an SLA is violated, the weight of the given tactic is changed to zero to yield precedence to other tactics. Hence, our example strategy depicted in Figure 6.8 first applies tactic *RemoveResources* because it has a weight of one and continues to apply this tactic until the strategy's objective is fulfilled or the weighting function assigns a weight of zero. Then, the strategy would continue with the *MigrateVM* tactic since it has a weight of 0.5.

6.3. Adaptation Framework Architecture and Implementation

In this section, we present a framework that implements the model-based adaptation control loop described in Chapter 4. Input to this framework is a Descartes Modeling Language (DML) instance. The framework interprets the adaptation process model (cf. Section 6.2) described as part of DML instance and applies the modeled adaptations on the resource landscape, application architecture, and deployment models (cf. Section 5.1). The output is an adapted model of the system with a configuration that solves the problem that triggered the adaptation process.

Figure 6.10 depicts the different software components of our framework. In the following, we walk through the architecture of our framework, starting at the ANALYZE phase and explaining the functionality of each component. The control flow between the components is depicted in Figure 6.11.

The `WCF` component forecasts future workload intensities based on the monitored workload data. We use the `ModelAdaptor` to apply the forecasts to the usage profile model of DML instance stored in the `ModelRepository`. Next, we query the `ModelAnalyzer` employing its online performance prediction techniques to evaluate the impact of the forecast workload changes on the system performance. In case the changes have a negative impact causing a performance problem, `WCF` triggers the `AdaptationController` to start an adaptation process.

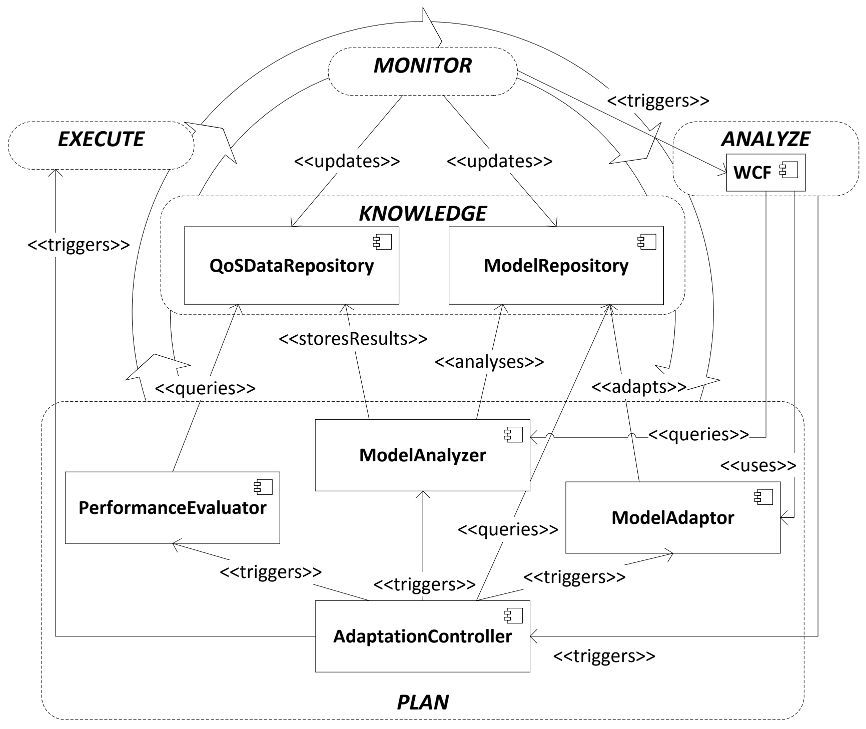


Figure 6.10.: Architectural overview of the adaptation framework.

The `AdaptationController` is the core component of our framework. The `AdaptationController` listens for events that trigger the PLAN phase to find a new system configuration that solves problems detected in the ANALYZE phase (cf. Section 4.1). When a problem has been signaled through an event from the ANALYZE phase, the `AdaptationController` launches the adaptation process to find a model configuration that resolves the identified issues. To start this process, the `AdaptationController` queries the `ModelRepository` for the strategy that has been designed to be triggered upon observation of the respective event. Next, it selects the tactic with the currently highest weight from the set of tactics assigned to the strategy (cf. Section 6.2). Then, the `AdaptationController` passes the selected tactic to the `ModelAdaptor`, which adapts DML instance in the `ModelRepository` according to the tactic's adaptation plan. The `ModelAdaptor` is the part of the adaptation framework that executes control flow elements like loops and branches and implements the execution of the modeled adaptation actions on the model level. In other words, the `ModelAdaptor` is the instance within our framework that ultimately defines the semantic of the modeled adaptation actions.

After the `ModelAdaptor` has applied the selected tactic, the `AdaptationController` calls the `ModelAnalyzer` to analyze the changed model instance. Depending on the performed adaptations, the `ModelAnalyzer` generates a performance query and triggers the online performance prediction process presented by Brosig (2014) and summarized in Section 2.2.4. The results of the model analysis are stored in the `QoSDataRepsitory`.

After model analysis, the `AdaptationController` triggers the `PerformanceEvaluator` to evaluate the impact of the tactic. The `PerformanceEvaluator` queries the metric values from the `QoSDataRepository` for the metrics that are referenced by the strategy's weighting function. Next, it uses the weighting function (cf. Section 6.2.5) to calculate the new weight for the tactic that has been applied last, and assigns this value to the tactic. Next, the `AdaptationController` decides if the detected problem has been solved or if further adaptations of the model with different tactics are required. To decide this, the `AdaptationController` checks if the metric values are within the range that has been specified by the strategy's objective. If a valid solution was found, the `AdaptationController` triggers the EXECUTE phase to adapt the real system. Otherwise, the process continues until a solution is found or a specific time limit is reached.

In the EXECUTE phase, system-specific adaptors replay the changes recorded by the `ModelAdaptor` on the actual system without considering adaptation steps that have been discarded in the model adaptation phase. These adaptors are the system-specific parts of our framework which must be implemented for the individual technology used in the adapted system, i.e., adaptors encapsulate the system-specific implementations of the modeled adaptation actions.

Implementation

All components—except of the workload classification and forecasting component `WCF`—are implemented in Java as OSGI components, using the Eclipse Equinox OSGi infrastructure. For code generation and editing of the model instances, we use the Eclipse Modeling Framework (EMF). All components are available as stand-alone Java applications or Eclipse plug-ins (Descartes Research Group, 2013).

To provide continuous workload forecasts, the `WCF` component is designed as a stand-alone Java application that is independent of the rest of the framework and is used in the ANALYZE phase for workload forecasting (cf. Chapter 7).

The `AdaptationController` is implemented as a Java thread that connects to the other components upon creation and then listens for triggering events from the ANALYZE phase. When triggered, it starts to adapt the model according to the given strategy until the problem that caused the event is solved or the period of time available for adaptation has elapsed.

To adapt a DML model instance, the `ModelAdaptor` employs the adapter classes generated by EMF and the EMF reflection API to interpret the adaptation plan and execute the defined adaptation actions. Furthermore, the `ModelAdaptor` uses the EMF Change Model API to track the changes performed on DML instance. This way, we can ensure the transaction semantics of tactics (cf. Section 6.2) and roll back the model to its previous state if the application of a tactic fails.

The `ModelRepository` provides access to the current DML instance. It also uses the adapter classes generated by EMF to query and inspect the model instances. The `ModelRepository` also uses EMF Change Model API to maintain a model history by tracing all changes that have been applied to the models. Thereby, we are able to switch back to a previous model state or use the trace to generate an adaptation plan that can be replayed on the real system in the EXECUTE phase.

The `QoSDataRepsitory` is implemented as an instance of a meta-model specified in EMF Ecore. In the context of this thesis, it manages the performance-relevant metrics measured in the system or predicted by the `ModelAnalyzer`.

The `ModelAnalyzer` implements the online performance prediction techniques described in Section 2.2.4. The actual implementation of this component is not in the scope of this thesis but presented in Brosig (2014).

The `PerformanceEvaluator` uses a special query language developed by Gorsler et al. (2014) to query the performance metrics relevant for evaluating the impact of an adaptation. The weighting functions for quantifying the adaptation impact are implemented in Java.

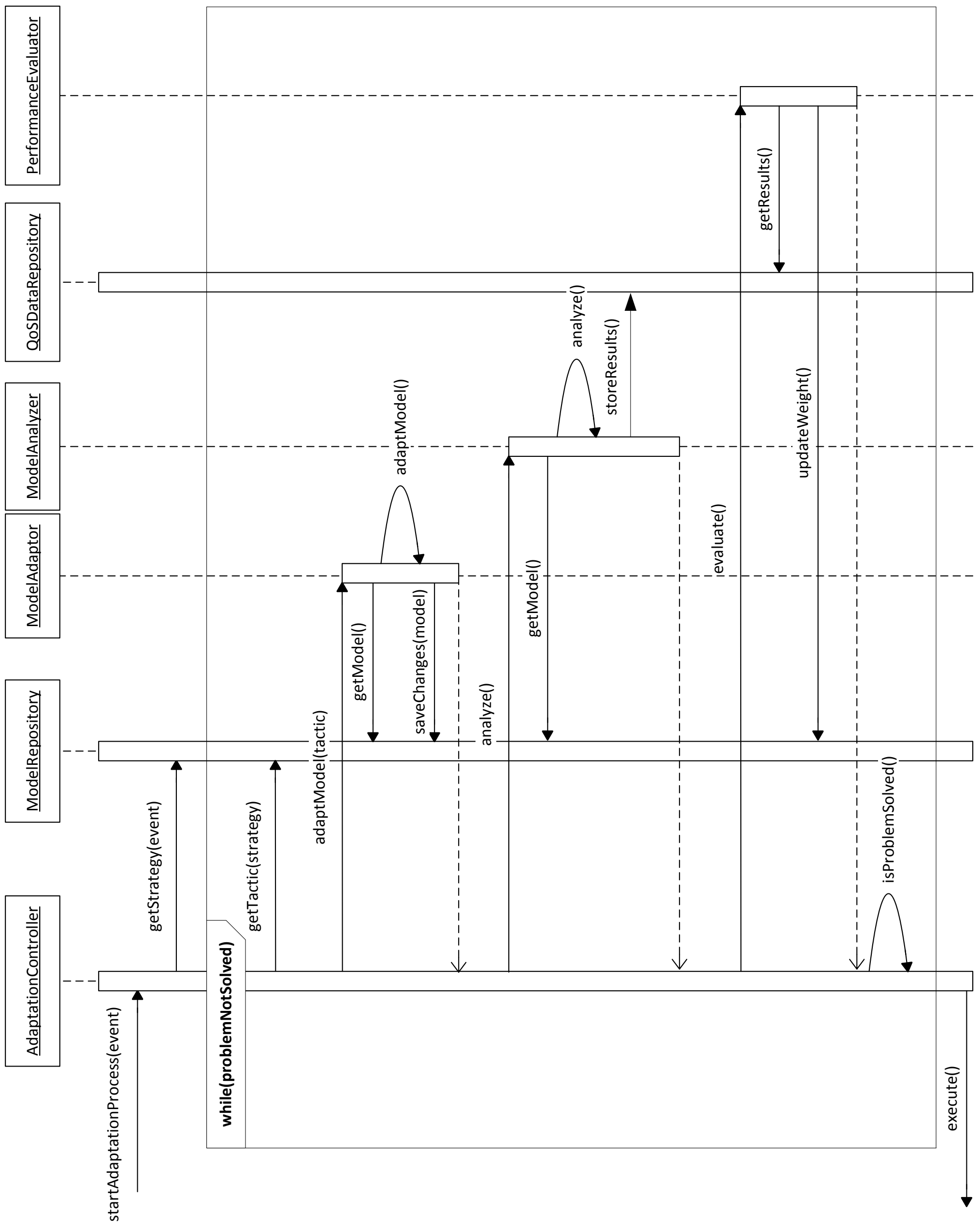


Figure 6.11.: UML sequence diagram of the interactions between the components of the adaptation framework.

6.4. Evaluation

In this section, we evaluate the adaptation process meta-model's generic and flexible formalism for modeling system adaptation. First, we assess our adaptation process meta-model and adaptation framework by comparing them with an existing classification of adaptation models (Vogel and Giese, 2012). Second, we demonstrate the efficiency and accuracy of our approach by comparing it with PerOpteryx (Martens et al., 2010), a framework for multi-objective software architecture optimization. Finally, we show the reusability of our meta-model in the context of SLAstic (van Hoorn, 2014), a framework for architecture-based online capacity management.

6.4.1. Comparing the Adaptation Process Meta-Model with Story Diagrams and Stitch

Vogel and Giese (2012) compiled an extensive list of requirements for adaptation models, considered as models that are used for planning and decision-making in adaptive systems. The list of Vogel and Giese comprises functional requirements (i.e., concepts required for analysis, decision-making, and planning) and non-functional requirements (requirements concerning the quality of the adaptation model). Furthermore, the list also includes framework requirements, i.e, requirements regarding the framework interpreting the adaptation model. Vogel and Giese use this list of requirements to evaluate two approaches, Stitch (Cheng and Garlan, 2012) and Story Diagrams (Fischer et al., 2000). We now use the same list to evaluate S/T/A, comparing it to Stitch and Story Diagrams. Table 6.1 shows how the three adaptation models support the different requirements, distinguishing between no support (-), partial support (P), and full support (F).

Concerning the functional language requirement *Goals* (LR-1), the OverallGoal and Objectives are the concepts used in S/T/A to specify the adaptation goal. Although this thesis is focused on performance management, S/T/A is designed to also support other *Quality Dimensions* (LR-2). With its weights of tactics and metrics, it also provides means to specify adaptation *Preferences* (LR-3). S/T/A has explicit *Access to Reflection Models* (LR-4) as it refers to the resource landscape and application architecture meta-model to query information about the current state of the system, which corresponds to Vogel and Giese's concept of a reflection model. Adaptations in S/T/A are triggered by *Events* (LR-5) that are emitted in the ANALYZE phase and the *Evaluation Conditions* (LR-6) can be described using Objectives. *Evaluation Results* (LR-7) are not an explicit part of the S/T/A adaptation language but they can be stored in and queried from the separate QoS data repository. *Adaptation Options* (LR-8) are not part of S/T/A but they can be specified as part of the adaptation points meta-model. The specification of *Adaptation Conditions* (LR-9) is supported at the level of Tactics and one can define different strategies tailored for special situations. *Adaptation Costs and Benefits* (LR-10) can be expressed by user-defined WeightingFunctions evaluating the impact and possible benefit of Tactics. Although we currently do not consider adaptation costs explicitly, costs can be integrated into the WeightingFunction in future work. Persisting the *History of Decisions* (LR-11) is not part of S/T/A but it is an important part of the adaptation framework interpreting S/T/A instances.

Our S/T/A adaptation language can be modularized along the three abstraction levels Strategies, Tactics, and Actions, but its scalability is limited to these concepts. Thus, it supports non-functional language requirements *Modularity, Abstractions and Scalability* (LR-12) only partially. In S/T/A, Actions are the only elements that actually affect the model or system, encapsulated in transaction-like Tactics. Thus, we clearly distinguish concepts that have *Side Effects* (LR-13). Furthermore, Tactics and Actions support *Parameters* (LR-14). However, as S/T/A is formally specified using a meta-model but has no

Table 6.1.: Comparison of the support of the language requirements (LR) and framework requirements (FR) of Stitch and Story Diagrams (cf. Vogel and Giese, 2012), extended with S/T/A: '-' indicates no support, 'P' indicates partial support, and 'F' indicates full support.

Functional Language Requirements

Req.	S/T/A	Stitch	SD	Req.	S/T/A	Stitch	SD	Req.	S/T/A	Stitch	SD
LR-1	F	-	F	LR-5	F	P	F	LR-9	F	F	F
LR-2	F	F	F	LR-6	F	F	F	LR-10	P	F	F
LR-3	F	F	F	LR-7	P	-	F	LR-11	P	P	F
LR-4	F	P	F	LR-8	P	F	F				

Non-functional Language Requirements

Req.	S/T/A	Stitch	SD	Req.	S/T/A	Stitch	SD	Req.	S/T/A	Stitch	SD
LR-12	P	P	F	LR-14	F	P	F	LR-16	P	P	P
LR-13	F	-	P	LR-15	P	-	P	LR-17	P	P	F

Framework Requirements

Req.	S/T/A	Stitch	SD	Req.	S/T/A	Stitch	SD	Req.	S/T/A	Stitch	SD
FR-1	F	P	F	FR-3	F	-	P	FR-5	F	-	F
FR-2	P	-	P	FR-4	F	-	F	FR-6	F	-	F

explicit formal definition of its semantics, the *Formality* requirement (LR-15) is fulfilled only partially. The S/T/A concepts are generally independent of an architecture-level performance model, but we can support *Reusability* (LR-16) only partially as the S/T/A model instances are coupled to concrete architecture-level performance model instances based on DML. Concerning the *Ease of Use* (LR-17), S/T/A requires understanding of modeling concepts (e.g., control flow diagrams), but no programming skills, etc.

Looking at the framework requirements (FR), our adaptation framework supports *Consistency* (FR-1), *Incrementality* (FR-2) and *Reversibility* (FR-3) because it works on the model level (where consistency checks can be executed) and employs techniques provided by the Eclipse Modeling Framework (EMF) to change models incrementally or reverse these changes, if necessary. Our adaptation framework also considers *Priorities* (FR-4) by assigning weights to tactics and metrics. As S/T/A is designed to be independent of the model analysis techniques, and adaptation strategies can be designed to suit certain time constraints, we can also support different *Time Scales* (FR-5). Our approach also supports *Flexibility* (FR-6) because the application of tactics varies during execution of the adaptation process.

As a result of comparing the support of functional and non-functional requirements we can rank our S/T/A adaptation language between Stitch, which focuses on system administration tasks, and the more general-purpose-like Story Diagrams. Note that this comparison focuses on the conceptual aspects of these languages and is based on information available in the literature (Vogel and Giese, 2012; Cheng and Garlan, 2012; Fischer et al., 2000). A more detailed assessment of technical aspects follows in the next section.

6.4.2. Comparing Accuracy and Efficiency Using PerOpteryx

Comparing approaches for self-adaptive systems in terms of properties like accuracy and efficiency is a big challenge (Cheng et al., 2009; de Lemos et al., 2011). The main reason is that to compare such approaches in a fair manner, the respective concepts and their implementations must be applicable in the same self-adaptive environment, ideally a benchmark for self-adaptive systems. Therefore, the compared approaches must provide mature tool support such that they can be applied outside of their target domain. Story Diagrams provide graphical editors to model and validate adaptation models and Stitch can be used in the *Rainbow* framework for self-adaptive systems (Garlan et al., 2004). However, the effort to adapt these approaches and their tools to one of our evaluation scenarios was too high. Furthermore, we did not have access to the environments in which Story Diagrams or Stitch have been applied. Thus, we decided to compare the accuracy of our approach with PerOpteryx, a framework for improving software architectures by automatically trading-off different design decisions (Martens et al., 2010). We integrated the concepts of strategies, tactics, and actions into the PerOpteryx framework. The idea is to execute the adaptation process modeled with S/T/A and compare it with PerOpteryx's optimization heuristics to show that our approach can find a suitable system configuration with sufficient accuracy and improved efficiency.

To improve the software architecture of a given architecture-level performance model, PerOpteryx searches the space of possible configurations of a given model instance for candidates that fit given optimization objectives. Therefore, PerOpteryx starts from the initial system configuration that has been modeled and generates new candidates along the degrees of freedom specified by the adaptation points meta-model. These new candidates are evaluated w.r.t. the given optimization objectives and a new optimization iteration starts with the best fitting candidates. Note that PerOpteryx is targeted at design-time optimization, i.e., there are no strict time constraints to evaluate a huge number of candidates. Although PerOpteryx implements heuristics encapsulating domain knowledge to reduce the number of candidates that are evaluated, PerOpteryx, with its implemented genetic algorithms, is not designed for use at run-time where quick results are required.

For our evaluation, we extended PerOpteryx and implemented a strategy and a set of tactics to constrain the generation of candidates (i.e., system configurations) to find a candidate that fulfills the objective of the strategy as quickly as possible. For our experiments, we used the same models and settings as in the Business Reporting System case study used to evaluate PerOpteryx (Martens et al., 2010). Figure 6.12 shows the output after 100 iterations with a population of 60. It depicts the set of Pareto-optimal candidates (marked by $\triangledown$) when trading-off response time (in seconds) vs. cost (an abstract unit used by PerOpteryx). We use this Pareto-optimal set of candidates as the baseline for the following accuracy and efficiency evaluation of our approach.

We assume that in this scenario the response time of the initial candidate (8.4 seconds) violates an SLA (response times < 5 seconds). This violation triggers our implemented strategy with the objective to adapt the system such that the response time is below five seconds. The strategy chooses a tactic from the given set of tactics $T = \{$*IncreaseResources*, *LoopIncreaseResources*, *BalanceLoad*$\}$ depicted in Figure 6.13. The *IncreaseResource* tactic implements one action, increasing the CPU capacity of the server with the highest utilization by 10% of its initial capacity. *LoopIncreaseResources* implements the same action but within a loop action repeating the *IncreaseCPU* action as often as specified by the loop's counter parameter. *BalanceLoad* migrates a software component from the server with the highest utilization to the server with the lowest. For the initial system state, we set the weights of the three tactics to (1.0, 0.0, 0.0). We then use the PerOpteryx framework to execute our strategy, choosing a tactic, executing it, and evaluating the resulting candidate. The evaluation reassigns weights to the tactics according to their effect. In this scenario, the WeightingFunction is kept simple, assigning a weight of zero to a tactic if it achieves no positive effect. The strategy continues applying the tactic with the highest weights until it reaches a state that fulfills the objective specified by the strategy. The resulting candidate of this process, i.e., system configuration that resolves the SLA violation, is depicted by the $*$ symbol in Figure 6.12.

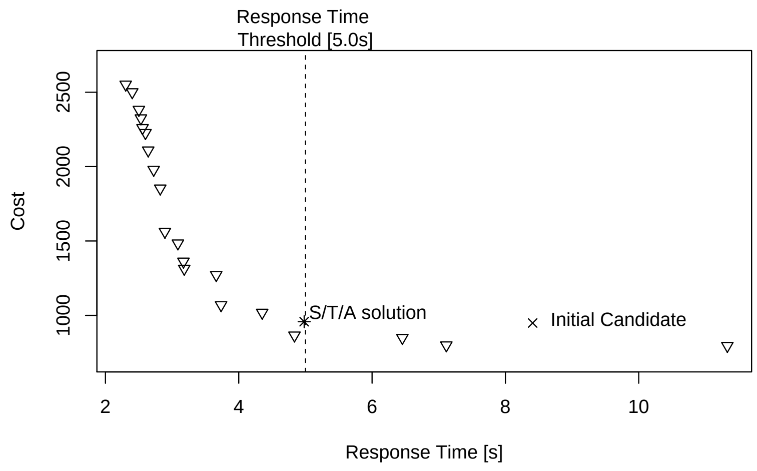


Figure 6.12.: Pareto-optimal candidates found by PerOpteryx (▽) and the candidate found when S/T/A is applied to guide the search process (∗).

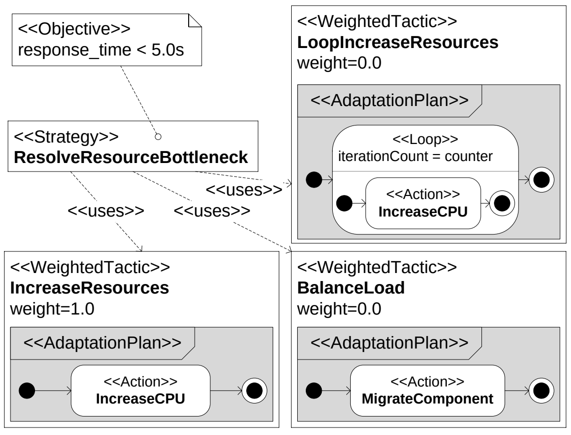


Figure 6.13.: Strategy, tactics, and actions used in the PerOpteryx scenario.

For our evaluation, we define *efficiency* as how much time is spent to find a system configuration that fulfills the given objectives. In model-based system adaptation, the time consumed to find a solution mainly depends on two factors. The first factor that can have a huge influence on the consumed time is the length of the model analysis. Coarse-grained analysis methods are quicker but produce less accurate results whereas simulation is much more accurate but also more expensive. To give an example, if we analyze our model with an LQN Solver (Franks et al., 2011), we obtain coarse-grained results within seconds whereas a simulation approach takes minutes, but provides much more detailed results. Thus, for systems of realistic size and complexity, a full-fledged search with PerOpteryx using simulative model analysis can typically take several hours or even up to several days. The second factor is how many iterations an adaptation approach needs to find a solution, i.e., how many model analysis steps the process requires. As model analysis can be expensive, reducing the number of iterations is essential for increasing the efficiency of our approach.

To evaluate the efficiency of S/T/A, we compare it to PerOpteryx based on these two factors. The results depicted in Figure 6.12 show that the configuration found using the

S/T/A model is not optimal. However, it fulfills the given objective and was found within eight iterations whereas the standard evolutionary search of PerOpteryx provides the first SLA-fulfilling candidate after ten iterations (cf. Figure 6.14). Thus, using S/T/A models can be more efficient to find a close to optimal solution than a full-fledged optimization approach. Furthermore, we see that five out of the eight iterations executed the *LoopIncreaseResources* tactic which executes two nested adaptation actions. In total, this results in saving five (out of thirteen) model analyses that would have been necessary without S/T/A. As a result, if strategies and tactics are specified to operate in an efficient manner, solutions can be found more efficiently. Note, that there are also further ways to speed up the analysis, for example, leveraging techniques for caching previous analysis steps.

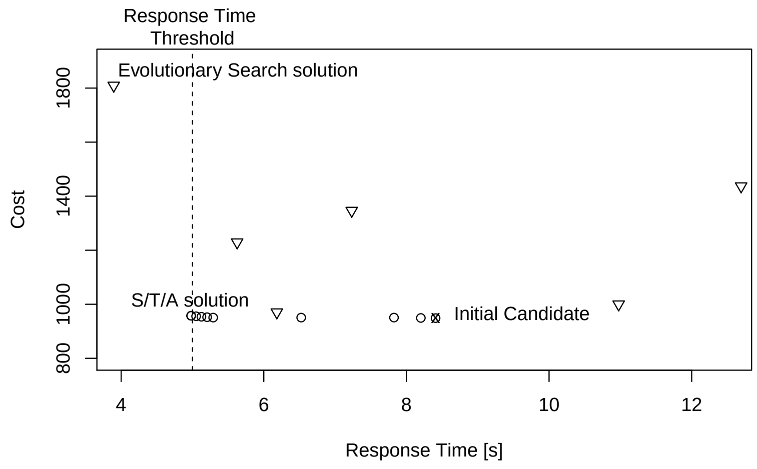


Figure 6.14.: Comparison of the candidates of the PerOpteryx evolutionary search (▽) with the candidates of S/T/A (○).

Finally, we emphasize that the contribution of our adaptation process meta-model is not a new optimization algorithm. Instead, our goal, which is illustrated with the described scenario, is to focus system adaptation processes such that a system configuration that fulfills given objectives is found as quickly as possible, i.e., the resulting system configuration must not necessarily be globally optimal. We can say that the accuracy with which a solution is found depends on the accuracy of the used architecture-level performance model and employed model analysis technique. If the architecture-level performance model reflects the system state within a given confidence level, the solutions found by S/T/A maintain the same accuracy. Furthermore, as the analysis time is a major factor influencing the efficiency of the adaptation process, one must carefully trade-off between accuracy and efficiency, i.e., if efficiency is key, one must sacrifice a certain amount of accuracy and vice versa.

6.4.3. Reusing Adaptation Plans in SLAstic

This section illustrates how the generic concepts of our adaptation process meta-model can be used in other adaptation scenarios based on architecture-level performance models. The results presented in the following have been created in cooperation with the University of Kiel, which also provided SLAstic, a framework for architecture-based online capacity management (van Hoorn, 2014; von Massow et al., 2011). SLAstic aims to increase the resource efficiency of distributed component-based software systems employing architectural run-time adaptations. SLAstic also relies on architectural models describing a system's QoS-relevant aspects and adaptation capabilities including assembly, deployment, QoS objectives, and reconfiguration capabilities. SLAstic's purpose is to determine required adaptations proactively in order to derive and execute appropriate adaptation plans. Its OverallGoal is to increase a system's efficiency, i.e., providing only as much capacity as required to satisfy the requested QoS objectives. In this scenario, we show how our adaptation process meta-model can be used to specify and execute architectural adaptation plans. An adaptation manager of the SLAstic framework executes these plans to perform the modeled actions with the goal to bring a real or simulated system from a current to a desired configuration.

The rest of this section presents some results of a lab experiment, employing SLAstic to control the capacity of a software system deployed on an Eucalyptus-based IaaS cloud environment, which is compatible with the Amazon Web Services (AWS) API (Eucalyptus Systems Inc., 2013). The five S/T/A actions depicted in Figure 6.15 correspond to the set of architectural run-time adaptation operations currently supported by the SLAstic framework: *allocate* and *deallocate* execution containers (i.e., physical or virtual servers), as well as *migrate*, *replicate*, and *dereplicate* a given software component to or, respectively, from a given execution container. Input parameters refer to types from the SLAstic meta-model. Note that these actions are the same regardless of whether SLAstic is connected to an IaaS environment or, for example, to a simulator for run-time adaptable PCM instances (von Massow et al., 2011).

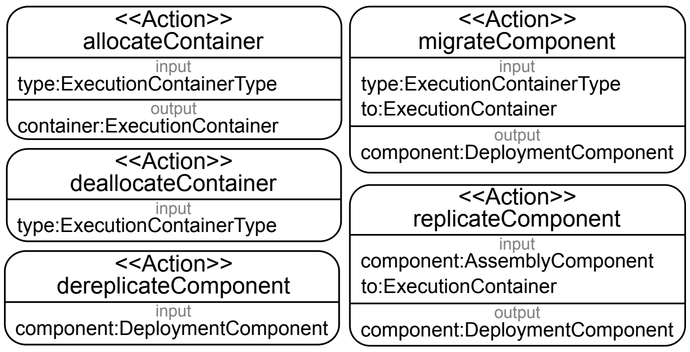


Figure 6.15.: Actions of the SLAstic online capacity management scenario.

In the cloud scenario, an adaptation manager receives pre-calculated AdaptationPlan instances and executes them by interacting with the AWS API and the allocated nodes according to the actions specified in Figure 6.15. To do so, SLAstic's adaptation manager maintains mappings between architectural model entities and their system-specific counterparts, e.g., mapping execution container types (ExecutionContainerType) to pairs of machine image and VM type, or mapping logical software component instances (AssemblyComponent) to the respective software artifacts to be deployed. For example, when executing the *allocateContainer* action, the adaptation manager calls the AWS service `run-instances` and performs subsequent initialization actions, such as starting an application server and system-level monitoring. As a second example, *replicateComponent* involves steps like deploying a software artifact (e.g., *.war* or *.ear* archives) corresponding to an assembly component into an application server on a previously allocated machine and updating the load-balancing configuration.

In our evaluation, we expose a Java-based application (JPetStore 5.0) to a probabilistic workload with varying intensity based on a 24-hour workload profile obtained from an industrial system. The profile was scaled to an experiment duration of 24 minutes plus 2 minutes cooldown. In this setting, we made the assumption that we have a good understanding of the correlation between application-level workload intensity—in this case, the number of requests to a software component per minute—and the CPU utilization.

For each software component, we defined a rule set specifying the number of component instances to be provided at certain workload intensity levels, e.g., five instances in periods with a workload intensity of 27,000 requests per minute. Deviations between the number of component instances specified in the rule set and the number of instances actually allocated, trigger the adaptation planner to create an adaptation plan with the goal to achieve the requested architectural configuration. This plan is then sent to the adaptation manager for execution.

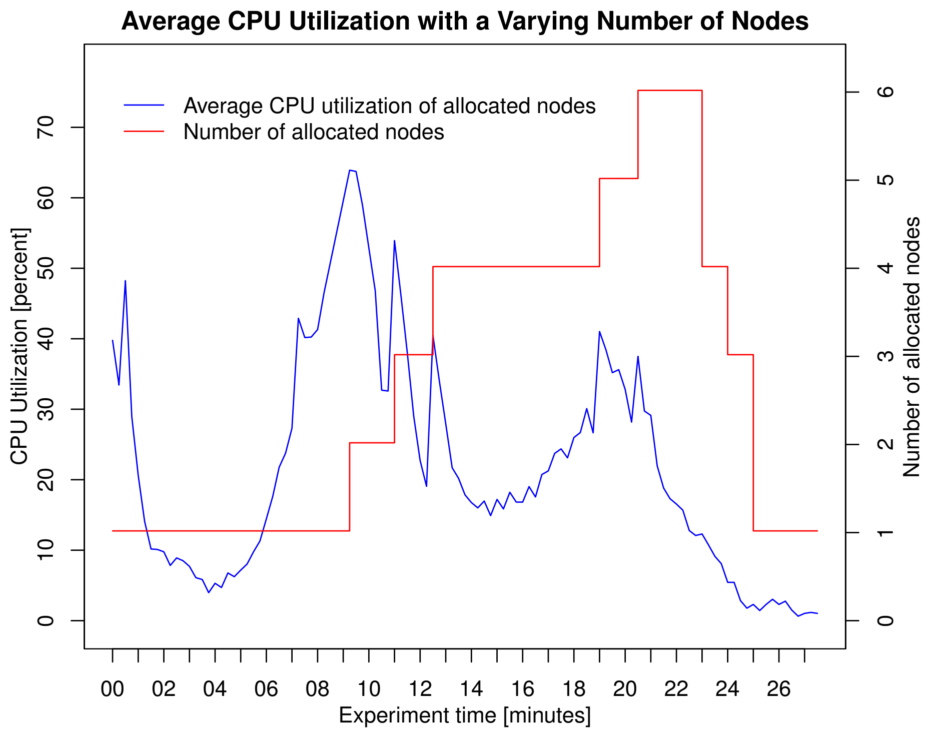


Figure 6.16.: Online capacity management executing modeled adaptation plans.

We executed the experiment with and without adaptation being enabled. In the latter scenario, a fixed number of 6 nodes was allocated throughout the entire experiment. Figure 6.16 shows the measured CPU utilization and the varying number of allocated nodes with adaptation enabled. The number of allocated nodes in this experiment varies between one and six. Comparing the results of both settings, the average CPU utilization increased with adaptation enabled, while (average) response times were very similar (5 ms measured at the application's entry points). More importantly, this scenario gives an example for decoupling adaptation planning from execution by using instances of our meta-model to describe pre-calculated adaptation plans. We successfully exchanged these adaptation plans between planning and executing parties to achieve a desired system configuration in a cloud scenario. This illustrates how the generic concepts of our adaptation process meta-model can be applied in other contexts, too.

6.5. Summary

In this chapter, we presented two meta-models, one to describe the adaptation points of a dynamic system and the second for modeling adaptation processes. Furthermore, we presented an architecture of an adaptation framework which implements the model-based adaptation control loop presented in Chapter 4, leveraging the novel features of DML as a basis for autonomic performance-aware and resource management. We evaluated our approach by comparing it with two other adaptation languages (Stitch and Story

Diagrams) and applying it in two different scenarios in the context of multi-objective software architecture optimization and online capacity management, respectively.

The results showed that our adaptation language and adaptation framework fulfill all essential requirements for adaptation models (Section 6.4.1). We also demonstrated how our adaptation language can improve the efficiency of system adaptation processes. However, we also observed that the efficiency highly depends on how strategies and tactics are modeled as well as on the desired accuracy of the solution. Vice versa, the accuracy of the the model analysis influences efficiency, since conduction detailed analysis can be time-consuming and depends on the accuracy of the underlying architecture-level performance model. Finally, we demonstrated the reusability of our adaptation process meta-model in the context of SLAstic, where pre-calculated architectural adaptation plans can be used to exchange information between the adaptation planning and the execution phase.

7. Self-Adaptive Workload Classification and Forecasting

A vital benefit of model-based adaptation approaches is that models can be leveraged to analyze the impact of changes in the system environment on the system's performance. However, a major challenge is to predict such changes in the environment accurately and continuously at system run-time. In the context of performance and resource management, a significant performance-influencing factor that is subject to change at run-time is the system's workload (cf. Chapter 4). Consequently, if we are able to predict the workload, i.e., the intensity with which the system is used, we can employ the model-based approach presented in the previous chapters to analyze and detect possible future performance problems and proactively adapt the system to maintain performance requirements.

In this chapter, we present an approach for self-adaptive Workload Classification and Forecasting (WCF) at run-time. We use well-established time series analysis techniques to identify the characteristics of workloads. Based on the identified characteristics, we select suitable forecasting methods to predict future workload intensities. These prediction results can then be used as input to model-based system adaptation approaches. Our approach automatically adapts the selection of the most effective forecasting method as the workload evolves during operation. Based on this mechanism, we are able to adjust the forecasting method at run-time to the given workload and forecasting requirements, thereby improving the forecasting accuracy without introducing significant additional computational overhead.

In Section 7.1, we explain the meanings of different terms we use in this chapter, discuss the characteristics workloads can expose, and present a survey of the most common time series analysis methods that we use for workload forecasting. In Section 7.2, we present our approach for self-adaptive Workload Classification and Forecasting (WCF), selecting from different time series analysis methods to achieve accurate forecasting results. Section 7.3 describes the implementation of our process, which is evaluated in Section 7.4. The approach presented in this chapter has been developed in collaboration with Herbst (2012) and published in (Herbst et al., 2014).

7.1. Time Series Analysis

Modern IT systems usually host multiple types of applications which offer different kinds of *software services*. These services are used by different users, which can be either humans or other systems. The invocation of a software service by a user is called *request*. Within the set of requests of a software service, we further distinguish different *request classes*. A request class is a category of requests characterized by statistically indistinguishable resource demands, i.e., requests of the same class consume the same amount of physical or virtual resources. For each request class, we can observe a distinct *workload* which we define as a time series of request arrival rates. In this thesis, a *time series* is a stochastic process, i.e., a collection of random variables $X = \{x_t : t \in T\}$, indexed by a totally ordered set T (the set of "time points"). Each x_t, a non-negative random number from the sample space $\mathbb{N}_0$, corresponds to the *request arrival rate*, the number of unique request arrivals of the same request class during the respective time interval $[t, t+1)$. The elapsed time between two time points in the time series is defined by a value and a time unit. Furthermore, the number of time series points that add up to the next upper time unit or another time period of interest is called the *frequency* of a time series. This is an important attribute of a time series which can be used as a starting point to search for seasonal patterns.

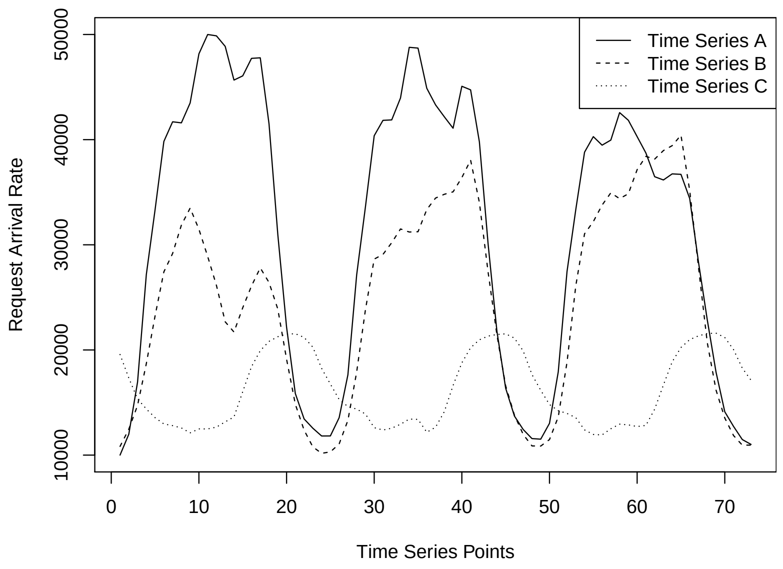


Figure 7.1.: Examples of three time series of arrival rates.

Figure 7.1 shows examples of three workloads over a period of three days. Each time series captures the number of request arrivals per hour for a given request class, i.e., the sampling interval in this example is an hour. One can also recognize a pattern repeating every 24 time series points. This is the frequency of these time series, which is 24 hours or one day. In time series A and B, we can also recognize a decreasing trend of the workload intensity behavior, respectively. The *workload intensity behavior* (WIB) is a description of the changes in the workload intensity over time including the shape of seasonal patterns and trends as well as the level of noise and bursts. Characterizing this intensity behavior is important to classify the workload. In the following Section 7.1.1, we first present the different characteristics that workload intensity behaviors can exhibit, which we leverage in this thesis for workload classification. Depending on the identified characteristics, different forecasting methods are applicable. In Section 7.1.2, we present a survey of the most common time series forecasting methods.

7.1.1. Workload Intensity Behavior Characteristics

According to the theory of time series analysis (Box et al., 2008; Hyndman, 2008; Shumway, 2011), a time series can be decomposed into the three components trend, season, and noise.

The relative weights and shapes of these components characterize the respective workload intensity behavior.

The *trend* component can be described by a monotonically increasing or decreasing function (in most cases a linear function) that can be approximated using common regression techniques. A break within the trend component is usually caused by system extrinsic events and therefore cannot be forecast based on historic observations. However, such breaks can be detected in retrospect, i.e., it is possible to estimate the likelihood of a change in the trend component by analyzing the durations of historic trends.

The *seasonal* component captures recurring patterns that are composed of at least one or more frequencies, e.g., daily, weekly, or monthly patterns. These frequencies can be identified by using a Fast Fourier Transformation (FFT) or by auto-correlation techniques.

The *noise* component is an unpredictable overlay of various frequencies with different amplitudes changing quickly due to random influences on the time series. The noise can be reduced by applying smoothing techniques like weighted moving average (WMA), by using lower sampling frequency, or by a low-pass filter that eliminates high frequencies. Finding a suitable trade-off between the amount of noise reduction and the respective potential loss of information can enhance the forecast accuracy.

Figure 7.2 illustrates the decomposition of a time series into the above mentioned components, as presented by Verbesselt et al. (2010). The first row shows the actual time series data. The second row contains detected (yearly) seasonal patterns, whereas the third row shows estimated trends and several breaks within these trends. The remainder depicted in the bottom row is the non-deterministic noise component computed by the difference between the original time series data and the sum of the trend and the seasonal components. The authors also offer an implementation of their approach for time series decomposition and detection of breaks in trends or seasonal components (Verbesselt, 2009).

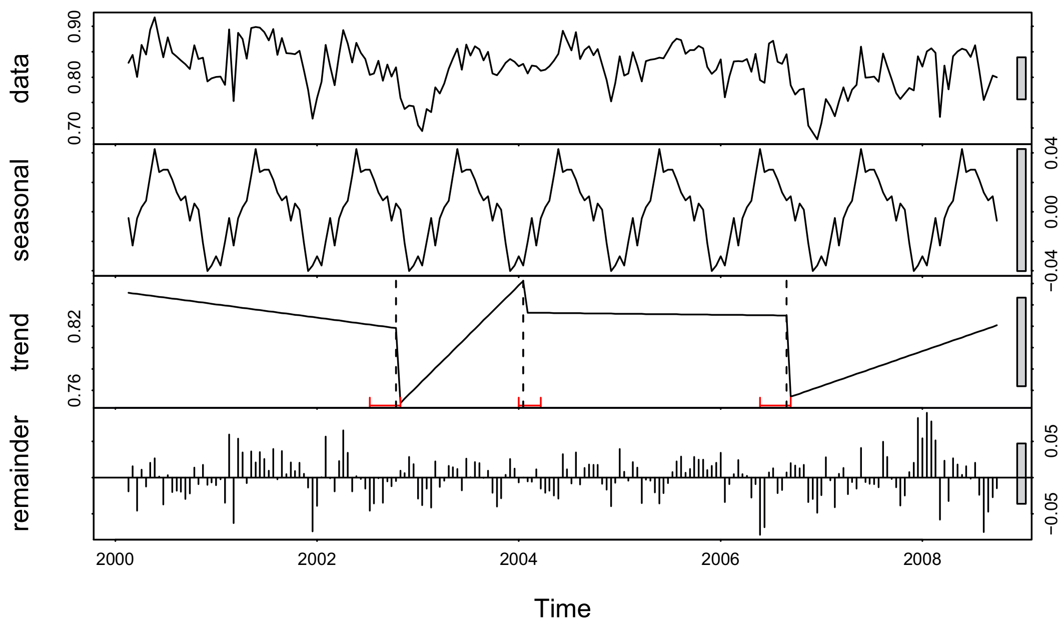


Figure 7.2.: Example time series decomposed into season, trend and noise components (cf. Verbesselt et al. (2010)).

The theory of time series analysis differentiates between static and dynamic stochastic processes. In static process models, it is assumed that the trend and seasonal components remain constant, whereas in dynamic (non-static) process models, these components change or develop over time and therefore have to be approximated periodically to achieve good forecast accuracy. Still, the trend and seasonal components are considered to be deterministic and the quality of their approximation is important input for forecasting methods. Depending on the applied stochastic model, the seasonal, trend and noise components can be considered either as multiplicative or additive.

A shortcoming of decomposing a time series into these components (e.g., by using the BFAST approach by Verbesselt et al. (2010)) is that the decomposition induces a high computational overhead. Hence, characteristics of a time series that can be computed with low overhead at run-time are crucial for efficient workload classification. In the following, we describe such characteristics.

The *burstiness* index is a measure for the weight of fluctuations (bursts) within the time series and calculated by the ratio of the maximum observed value to the median within a time frame.

The *length* of the time series mainly influences the accuracy of approximations for the above mentioned components and limits the space of applicable forecasting methods.

The number of either increasing or decreasing *consecutive monotonic values* within a time frame indirectly characterizes the influence of the noise and seasonal components. A small value can be interpreted as a sign of a high noise level and a hint to apply a time series smoothing technique.

The *maximum*, *median*, and *quartiles* are important indicators of the distribution of the time series data and can be unified in the *quartile dispersion coefficient* (QDC), defined as the distance of the quartiles divided by the median value.

The standard deviation and the mean value are combined in the *coefficient of variation* (COV), which characterizes the dispersion of the time series value distribution as a dimensionless quantity.

Absolute *positivity* of a time series is an important characteristic because intervals containing negative or zero values can influence the forecast accuracy and even the applicability of certain forecasting methods. As arrival rates cannot be negative by nature, a time series that is not absolutely positive should be subjected to a simple filter eliminating the zero or negative values or it should be analyzed using specialized forecasting methods.

The *relative gradient* is defined as the relation of the absolute gradient of the latest quarter of a time series period to the median of this quarter. It captures the steepness of the latest quarter of the period. A positive relative gradient shows that the last quarter of the period changed less than the median value, a negative value indicates a steep section within the time series (e.g., the limb of a seasonal pattern).

As mentioned previously, the *frequency* of a time series represents the number of time series values that form a *period* of interest (in most cases simply the next bigger time-unit). These values are an important input as they are used as starting points for the search for seasonal patterns.

7.1.2. Survey of Forecasting Methods

In this section, we compare the most common forecasting approaches in the field of time series analysis. In a brief summary, we highlight their requirements, advantages and disadvantages, based on (Box et al., 2008; Hyndman, 2008; Hyndman and Khandakar, 2008; Shumway, 2011; De Livera et al., 2011; Hyndman et al., 2002; Shenstone and Hyndman, 2005). All presented forecasting methods have been implemented in the R forecast package by Hyndman (2009) and documented in (Hyndman and Khandakar, 2008). The implemented forecasting methods are based either on the innovations state-space approach (Hyndman, 2008) or on the auto-regressive integrated moving averages (ARIMA) approach for stochastic process modeling (Box et al., 2008). These two general approaches for stochastic process modeling have common aspects, but are not identical as in both cases there exist model instances that have no counterpart in the other approach. Both have different strengths and weaknesses as discussed, e.g., by Hyndman and Khandakar (2008). Note that further forecasting methods, like the Theta model (Assimakopoulos and Nikolopoulos, 2000) or the Holt-Winters approach (Goodwin, 2010), are covered in our presented list as a special case of exponential smoothing.

In the following sections, we discuss the properties of different forecasting methods we consider for our online workload classification and forecasting approach. Table 7.1, summarizes the presented forecasting methods and their most important properties, the requirements to apply the respective method as well as its benefits and shortcomings. Concerning the computational complexity, the description in common O-notation is not feasible in most cases as the shape of seasonal patterns contained in the time series data as well as the used optimization thresholds during a model fitting procedure strongly influence the computational overhead. Therefore, we evaluated the computational complexity of the individual methods by running experiments with a representative amount of method executions on a machine with an Intel Core i7 CPU (2.7 GHz). The forecasting methods can only utilize a single core, as multi-threading is not yet fully supported by the existing implementations.

7.1.2.1. Naive Forecast

The underlying assumption of the *naive* forecasting method is that the next observation is most likely the same as the one that was currently observed. This method is usually employed as the reference method for comparing forecast approaches (Hyndman and Koehler, 2006). It can be combined with a random-walk factor or a drift. This method induces no computational overhead besides the calculation of a confidence interval and requires only a single time series point to be applicable. The *naive* method is identical to the moving averages method with a sliding window of size one.

7.1.2.2. Moving Averages

The *moving averages* (MA) method computes the unweighted mean of the previous n time series points as the point forecast for the next interval. As the time series develops, the sliding window of n time series points and the respective mean develops, too. Compared to the *naive* method, this method is able to smooth out a certain noise level by averaging over the sliding window. The computational overhead is very low, i.e., in $O(log(n))$.

7.1.2.3. Simple Exponential Smoothing

The *simple exponential smoothing* (SES) method extends the MA approach by weighting the more recent values of the sliding window with exponentially higher factors. In the first step of SES, the parameters of the used exponential function are estimated for the given time series data by employing an iterative optimization process. In the second step, point forecasts and confidence intervals are computed iteratively. This method smooths out a certain noise level and reacts on the influences of trend or seasonal patterns due to the exponential weighting in a more flexible manner than MA. But still, as this method inherently damps any changes in the values, it does not extrapolate trends or other developments due to seasonal patterns. Experiments showed that the SES method returns a result below 80 milliseconds when applied on less than 100 values. This computational

Table 7.1.: Summary of forecasting methods based on time series analysis.

Name	Description	Forecast Horizon	Requirements	Overhead	Benefits	Shortcomings	Opt. Application Scenario
Naive Forecast, comparable to ARIMA100	The naive forecast considers only the value of the most recent observation assuming that this value has the highest probability to be observed next.	**very short** term forecast (1-2 points)	single observation	**nearly none**, O(1)	no historic data required, reference method	naive, i.e. no value for proactive resource management	constant arrival rates
Moving Average (MA),comparable to ARIMA001	Calculation of an arithmetic mean within a sliding window of the *n* most recent observations.	**very short** term forecast (1-2 points)	two observations	**very low**, O(log(n))	simplicity	sensitive to trends and seasonal patterns	constant arrival rates with low noise
Simple Exponential Smoothing (SES), comparable to ARIMA011	Similar to MA but uses an exponential weighting function to assign higher weights to more recent values. 1st step: estimation of parameters for weights/exp. function 2nd step: calculation of weighted averages as point forecasts	**short** term forecast (< 5 points)	small number of historic observations (> 5)	**low**, below 80 ms for less than 100 values	more flexible reaction on trends or other developments than MA	no seasonal component, only damping and no interpolation	time series with little noise and changes within trend, but no seasonal behavior
Cubic Smoothing Splines (CS), comparable to ARIMA022	Cubic splines are fitted to the univariate time series data to obtain a trend estimate and linear forecast function. Prediction intervals are constructed by use of a likelihood approach for estimation of smoothing parameters. The CS method can be mapped to an ARIMA022 stochastic process model with a restricted parameter space.	**short** term forecast (< 5 points)	small number of historic observations (> 5)	**low,** below 100 ms for less than 30 values (more values do not sig. improve accuracy)	extrapolation of the trend by a linear forecast function	sensitive to seasonal patterns (steep edges), negative forecast values possible	strong trends, but minor seasonal behavior, low noise level
Auto-Regressive Integrated Moving Averages (ARIMA101)	ARIMA101 is an ARIMA stochastic process model instance parameterized with $p = 1$ as order of *AR(p)* process, $d = 0$ as order of integration *I(d)*, $q = 1$ as order of *MA(q)* process. In this case of stationary stochastic processes, (no integration) and no seasonality is considered.	**short** term forecast (< 5 points)	small number of historic observations (> 5)	**low,** below 70 ms for less than 100 values	trend estimation, (more careful than CS), fast	no seasonal component modeled, only positive time series	time series with some noise and changes within trend, but no seasonal behavior
Croston's Method (Intermittent demand forecasting)	Decomposition of the time series that contains zero values into two separate sequences: a non-zero valued time series and a second sequence that contains the time intervals of zero values. Independent forecast using SES and combination of the two independent forecasts. No confidence intervals are computed due to not consistent underlying stochastic model.	**short** term forecast (< 5 points)	small number of historic observations, containing zero values (> 10)	**low**, below avg. 100 ms for less than 100 values	handles time series containing zeroes as values	no seasonal component, no confidence intervals due to no underlying stochastic model	zero valued periods, active periods with trends, no strong seasonal comp., low noise
Extended Exponential Smoothing (ETS) (Innovation state space stochastic model framework)	1st step: Model estimation. Noise, trend and season components are either additive (A), or multiplicative (M) or not modeled (N). 2nd step: Estimation of parameters for explicit noise, trend and seasonal components. 3rd step: Calculation of point forecasts for level, trend and season components independently using SES and combination of results.	**medium to long** term forecast (> 30 points)	at least 3 periods in time series data with sufficient frequency	**high**, up to 15 seconds for less than 200 values, high variability in computation time	capturing explicit noise, trend and seasonal component in a multiplicative or additive innovation state space model	only non-negative time series	times series with clear and simple trend and seasonal component, moderate noise level
tBATS (Trigonometric seasonal model, Box-Cox transformation, ARMA errors, trend + seasonal components)	The tBATS stochastic process modeling framework of innovations state space approach models complex seasonal time series (multiple/high frequency/non-integer seasonality) and uses Box-Cox transformation, Fourier representations with time varying coefficients and ARMA error correction. Trigonometric formulation of complex seasonal time series patterns enables identification using FFT and time series decomposition. Improved computational overhead using a new method for maximum-likelihood estimations.	**medium to long** term forecast (> 5 points)	at least 3 periods in time series data with an adequate frequency	**high**, up to 18 seconds less than 200 values, high variability in computation time	modeling capability of complex, non-integer and overlaying seasonal patterns and trends	only non-negative time series, complex process for time series modeling and decomposition	times series with one or more clear but possibly complex trend and seasonal components, only moderate noise level
ARIMA (Auto-regressive integrated moving averages stochastic process model framework with seasonality)	The automated ARIMA model selection process of the R forecasting package starts with a complex estimation of an appropriate ARIMA$(p, d, q)(P, D, Q)_m$ model by using unit-root tests and an information criterions (like the AIC) in combination with a step-wise procedure for traversing a relevant model space. The selected ARIMA model is then fitted to the data to provide point forecasts and confidence intervals.	**medium to long** term forecast (> 5 points)	at least 3 periods in time series data with an adequate frequency	**very high**, up to 50 seconds for less than 200 values, high variability in computation time	capturing noise, trend and season component in (multiplicative or additive) model, achieves close confidence intervals	only non-negative time series, complex model estimation	times series with clear seasonal component (constant frequency), moderate noise level

overhead is mainly induced by the parameter estimation step. SES is suitable for short-term forecasts and can be applied on a time series of small size. In the SES method no trend or seasonal component is considered. This is covered in the ETS method. The SES method is equal to an application of the ARIMA($(p, d, q) = (0, 1, 1)$), cf. Section 7.1.2.9.

7.1.2.4. Cubic Smoothing Splines

As demonstrated and discussed by Hyndman et al. (2002), *cubic smoothing splines* (CS) can be fitted to univariate time series data to obtain a linear forecast function that estimates the trend of the time series. The smoothing parameters are estimated using a likelihood approach, enabling the construction of confidence intervals. The authors define this approach is a special case of the ARIMA($(p, d, q) = (0, 2, 2)$) model (cf. Section 7.1.2.9) with a restricted parameter set. They demonstrate that this restriction does not affect the forecast accuracy. This approach tends to estimate trends better than the SES method, but seasonal patterns cannot be captured. Also, this method sometimes overestimates a trend in steep parts of a time series. The computational overhead remains below 100 milliseconds when applied on 30 values. We have observed that the computation time rises for more values without an observable improvement in the forecast accuracy.

7.1.2.5. ARIMA(1,0,1) Stochastic Process Model

The ARIMA($(p, d, q) = (1, 0, 1)$) model is an instance of the Auto-Regressive Integrated Moving Averages (ARIMA) framework described in Section 7.1.2.9. It assumes a stationary stochastic process (constant mean value) as it does not make use of the integration ($I(d)$) part (unlike SES and CS). Therefore, it can also be considered as an ARMA($(p, q) = (1, 1)$) process model (Shumway, 2011). This method is not as sensitive to steep parts in a time series as the CS method. Like in SES and CS, the application of the auto-regressive moving average includes a parameter estimation that induces the major computational effort. Our experiments showed that this method returns results below 70 milliseconds when applied on the last 100 values.

7.1.2.6. Croston's Method for Intermittent Time Series

Croston's method, as presented by Shenstone and Hyndman (2005), is specialized for forecasting of intermittent time series. In contrast to other methods, it is applicable to time series that contain zero values. Internally, the original time series is decomposed into a time series without zero values and a second time series that captures durations of zero valued intervals. These two time series are then independently forecast using the SES method and then unified. As this method uses no underlying stochastic model, confidence intervals cannot be computed. This method is based on SES, and the computational overhead is slightly higher with 100 milliseconds computation time on 100 values.

7.1.2.7. Extended Exponential Smoothing

Hyndman (2008) and Hyndman and Khandakar (2008) present an *extended exponential smoothing* or *Error-Trend-Seasonal* (ETS) method. This method is based on the innovations state space approach by Hyndman (2008). ETS explicitly models an error (or noise), a seasonal, and a trend component in individual SES equations that are combined in the final forecast result in an either additive or multiplicative (or neglected) manner. In addition, damping the influence of one of these components is possible. The forecast process starts with the selection of an optimized model, before the parameters of the single SES equations are estimated. Having the model and the parameters adapted to the time series, point forecasts and confidence intervals are computed. This method is able to detect and capture sinus like seasonal patterns that are contained at least three times in the time series. In such cases, the ETS has a computation time of 15 seconds on 200 time series values. In case of more complex patterns, this method is multiple times faster, but unable to detect the pattern, resulting in worse forecast accuracy.

7.1.2.8. tBATS Innovations State Space Modeling Framework

The *tBATS* innovations state space modeling framework presented by De Livera et al. (2011) has recently been integrated into the R forecasting package. It extends the ETS state space model for a better handling of more complex seasonal effects by making use of a trigonometric representation of seasonal components based on Fourier transformations, by the incorporation of Box-Cox transformations, and by use of ARMA error correction. tBATS relies on a method that reduces the computational burden of the maximum likelihood estimation. In experiments, processing times of up to 18 seconds have been observed on 200 values, but in several cases a processing time of five to seven seconds still resulted in appropriate forecast accuracy.

7.1.2.9. ARIMA Stochastic Process Modeling Framework

The *ARIMA* (Auto-Regressive Integrated Moving Averages) stochastic process modeling framework is presented in Box et al. (2008). The ARIMA model space is defined by seven parameters (p, d, q) and $(P, D, Q)_m$, where the first tuple defines the model's trend and noise component. The non-negative integer parameters p, d, and q refer to the order of the autoregressive (AR), integrated (I), and moving average (MA) parts of the model, respectively. The second tuple $(P, D, Q)_m$ is optional and defines a model for the seasonal component. The parameter m stands for the frequency of the seasonality. The model selection is a difficult process that can be realized via space limitation and intelligent model space traversion using different unit-root tests (KPSS, HEGY or Canova-Hansen) and Akiake's information criterion (AIC). Hyndman and Khandakar (2008) propose a process for automated model selection that is implemented in the `auto.arima()` function of the R forecast package. A selected ARIMA model is then fitted to the time series data to compute point forecasts and confidence intervals. The model selection and further fitting induces a high computational overhead of up to 50 seconds on 200 values with a high variance. The reason is that the model selection process depends on the actual data itself and not only on the quantity of the data. Experiments also showed that this ARIMA approach achieves better confidence intervals than the tBATS approach in most cases.

7.2. Workload Classification and Forecasting

In this section, we present a self-adaptive Workload Classification and Forecasting (WCF) process that can be used for selecting suitable forecasting methods at run-time. Over time, a workload intensity behavior may change and develop in a way that affects its characteristics, i.e., the workload intensity behavior class is not fixed and needs to be updated periodically. Therefore, to take such changes into account, our classification process must be self-adaptive, i.e., it must consider changes of the workload intensity behavior and of given forecasting objectives and automatically adapt the selection of appropriate forecasting methods.

An overview of the Workload Classification and Forecasting (WCF) process is sketched in Figure 7.3. Input of the WCF process is a trace of a workload intensity behavior, a set of forecasting objectives, and possible feedback about the accuracy of the previous forecast. Essentially, we distinguish two phases in this process, the classification phase and the forecasting phase. In the *classification phase*, we extract the characteristics of a given workload intensity behavior trace. Based on the identified characteristics, we use a decision

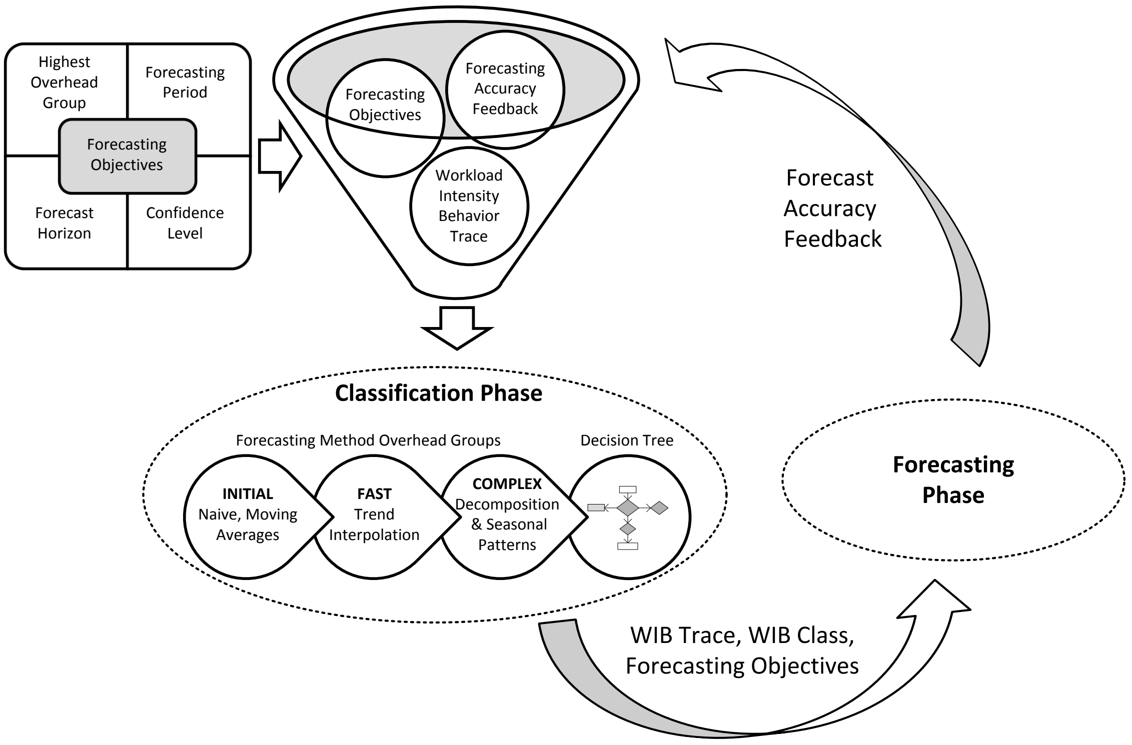

Highest Overhead Group
Forecasting Period
Forecasting Objectives
Forecast Horizon
Confidence Level
Forecasting Objectives
Forecasting Accuracy Feedback
Workload Intensity Behavior Trace
Forecast Accuracy Feedback
Classification Phase
Forecasting Method Overhead Groups
Decision Tree
INITIAL Naive, Moving Averages
FAST Trend Interpolation
COMPLEX Decomposition & Seasonal Patterns
Forecasting Phase
WIB Trace, WIB Class, Forecasting Objectives

Figure 7.3.: Overview of the workload classification and forecasting (WCF) process.

tree (cf. Section 7.2.6) to classify the workload intensity behavior and to select a suitable forecasting method. The assignment of the forecasting method also defines the class of the workload intensity behavior (WIB class). Then, in the *forecasting phase*, we apply the assigned forecasting method according to the given forecasting objectives. Our process also supports evaluating the accuracy of forecasts and uses the evaluation results as feedback for the next classification cycle. The classification and forecasting phases are executed iteratively, with a frequency that can be specified by the user. Thereby, our approach is able to continuously adapt the classification of the workload intensity behavior based on the evolution of the workload over time. However, the two phases must not necessarily follow each other iteratively. They can also be executed in parallel, i.e., the classification of the workload intensity behavior may be updated while also forecasting is performed.

In the following, we explain in detail the individual parts of our self-adaptive WCF approach.

7.2.1. Forecasting Objectives

Time series forecasts can be used for a variety of purposes from short term proactive resource provisioning to long term capacity planning. Depending on the purpose, different forecasting methods with different configurations may provide better or quicker results. Our approach explicitly takes into account the forecasting objectives guiding the forecast result processing to control the forecasting overhead. Our approach supports specifying the following parameters as forecasting objectives:

1. The *Highest Overhead Group* parameter is a value in the interval $[1, 4]$ that specifies the highest overhead group that the WCF approach can choose from. This value refers to the overhead groups we introduce in Section 7.2.2.

2. The *Forecast Horizon* parameter is a tuple of two positive integer values quantifying the number of time series points to be forecast. The first value of the tuple defines

the *Start Horizon* and defines the number of time series points to be forecast at the beginning. The second value defines the *Maximum Horizon*, i.e., the maximum number of time series points to be forecast. The start value can then be dynamically increased stepwise up to the maximum value. This is necessary because of the significant differences of the forecasting methods in terms of processing times and feasibility for long term forecasts.

3. The *Confidence Level* parameter can be a value $\alpha \in [0, 100)$ defining the percentage of how many of the calculated confidence intervals have to include the forecast mean value.

4. The *Forecasting Period* parameter is a positive integer i specifying how often in terms of the number of time series points a forecasting method is executed. For example, if $i = 1$ a forecast is executed for every new time series point that is added to the time series. If $i = 10$, the forecast is executed for every tenth time series point.

By specifying these parameters according to the characteristics of the considered forecasting scenario, one can influence the classification and forecasting process in finding a suitable forecasting method for the given workload intensity behavior. For example, assume an offline capacity planning scenario, where there is sufficient time to analyze the workload intensity behavior and make forecasts for a relatively long time period. In such a case, one can set the objectives to the highest overhead group and to the desired maximum forecast horizon. Furthermore, if high forecasting confidence is required, one could also set $\alpha = 95$ to specify a high confidence level.

7.2.2. Forecasting Methods Overhead Groups

In Section 7.1.2, we have presented a survey of different time series forecasting methods and discussed their computational overhead. In the following, we divide these forecasting methods into the following groups.

Group 1 – Negligible Overhead
This group contains methods with almost no overhead, such as the *Moving Average (MA)* and the *Naive* forecasting methods.

Group 2 – Low Overhead
This group contains methods with low overhead such as the fast forecasting methods *Simple Exponential Smoothing (SES)*, *Cubic Spline Interpolation (CS)*, the predefined *ARIMA101* model, and the specialized *Croston's Method* for intermittent time series. The processing times of forecasting methods in this group are below 100 ms for a maximum of 100 time series points.

Group 3 – Medium Overhead
This group stands for medium overheads and contains the forecasting methods *Extended Exponential Smoothing (ETS)* and *tBATS*. The processing times are below 30 seconds for less than 200 time series points.

Group 4 – High Overhead
This group stands for high overheads and contains again *tBATS* and additionally the *ARIMA* forecasting framework with automatic selection of an optimal *ARIMA* model. The processing times for methods in this group are below 60 seconds for less than 200 time series points.

7.2.3. Forecasting Methods Partitions

In addition to groups of computational overhead, we distinguish three major partitions of forecasting methods. Each partition contains forecasting methods that are applicable in different situations during the run-time workload classification and forecasting process. When a time series of request arrival rates is added for classification and no historic data of its workload intensity behavior is available yet, the only option is to apply fast and naive forecasting methods to obtain a basic forecast. At a later point in time, when more observations become available, it may be useful to apply methods that can interpolate the trends within the time series. And again at a later point in time, when about three periods within the time series have already been observed, it is possible to detect deterministic seasonal patterns by using complex time series analysis, decomposition and forecasting methods.

Therefore, our classification process distinguishes three partitions according to the amount of historic data available in the time series: an *initial* partition, a *fast* partition, and a *complex* partition. These partitions are related to the overhead groups of the forecasting methods. Having a short time series, i.e., for the initial partition, only forecasting methods in the *overhead group 1* can be applied. A medium length of a time series may allow application of methods contained in the *overhead group 2* and a long time series (complex partition) enables the use of methods in *overhead groups 3 and 4*. The two thresholds that define when a time series is considered as short, medium or long can be configured as parameters of the classification process. Based on experience gained from experiments, we recommend to set the threshold for the transition from initial to fast to a value $\tau \in [5, \frac{p}{2}]$ observations (five being the minimal amount of observations needed for Cubic Spline Interpolation and p being the length of a period in time series points). The fast to complex threshold should be set to a value $\tau \geq 3p$ because most methods in the respective overhead group need at least three pattern occurrences to identify them.

7.2.4. Evaluating Forecasting Accuracy

In an online scenario, workload intensity behaviors are not stationary, i.e., their characteristics can change over time. Accordingly, the most appropriate forecasting method can also change and must be adapted at run-time. It is the responsibility of the classification process to detect such changes and adjust the WIB class to the forecasting method that yields the most accurate results for the given workload intensity behavior considering the forecasting objectives.

To evaluate the accuracy of a forecasting method, a number of metrics assessing the differences between forecast results and corresponding observations have been proposed (cf. Hyndman and Koehler (2006) for an overview). The *Mean Absolute Scaled Error* (MASE) is usually the metric of choice that enables consistent comparisons of forecasting methods across different data samples. The MASE metric for an interval $[m, n] \in T$ is defined as follows:

$$MASE[m,n] = \frac{1}{n-m+1}\sum_{t=m}^{n}\left(\left|\frac{error_t}{b_{[m,n]}}\right|\right),$$

whereas $error_t$ is defined as

$$error_t = forecast_t - observation_t,\ t \in [m,n]$$

and $b_{[m,n]}$ as the average change within the observation

$$b_{[m,n]} = \frac{1}{n-m} \cdot \sum_{i=m+1}^{n} \left|observation_i - observation_{i-1}\right|.$$

Essentially, the MASE metric compares the forecast accuracy with the accuracy of the naive forecasting method. If the MASE value is close to one or even bigger, the computed forecast results are of no value because their accuracy is equal or even worse than using the naive forecasting method. This means that one could use monitoring data for resource provisioning, directly. In turn, the closer the metric value is to zero, the better the accuracy of the selected forecasting method.

In our classification process, we select the forecasting method providing the lowest MASE value for the given workload intensity behavior. The best point in time during the WCF process to precisely calculate the MASE metric is just before the next forecast execution is triggered, i.e., when both the observation values as well as the forecast results are available. The results are used as forecast accuracy feedback in the classification phase. However, in an online scenario where we need forecast results for proactive resource provisioning, no observations are available to assess the quality of the forecasts. In such situations when no observations are available, the MASE metric can also be used to assess the accuracy of different forecasting methods by comparing them with the naive forecast. Thus, we execute two or more forecasting methods in parallel during the forecasting phase and compare their accuracy in configurable intervals using the MASE metric to ensure that the currently chosen WIB class still produces the most accurate results compared to other approaches. This comparison of forecasting methods is also triggered when the evaluation of forecast results with observations seems implausible or shows low accuracy ($MASE[m, n] \geq 1$).

7.2.5. Non-Absolutely Positive Workloads

The workloads considered in this thesis contain time series of request arrival rates. These time series are non-negative, as there cannot be a negative number of requests per time period. However, the time series might contain zero values, as there might exist time periods where no requests arrive. The problem is that the majority of forecasting methods assume absolutely positive time series as input (cf. Section 7.1). Such forecasting methods are not numerically stable, i.e., they interrupt after a division by zero and thus cannot return a forecast result. Therefore, we need to eliminate zeros from the time series before passing it to the forecasting method.

However, if zero values appear regularly in the time series, it is better to use Croston's forecasting method for intermittent demands which is developed for such time series. This method decomposes the time series into two different time series. A strictly positive time series and another time series containing the period duration when the time series was zero. The forecast is then executed independently for both time series and combined later on.

However, the classification of a workload intensity behavior would be highly sensitive to zero values if only a few observations of zero values immediately caused a switch to the Croston's forecasting method. To configure this sensitivity, we introduced a threshold for the rate of zero values. We recommended setting this threshold to a reasonable value between 20% and 40%.

7.2.6. Decision Tree

The decision tree depicted in Figure 7.4 is the core element for selecting a suitable forecasting method. To determine a suitable forecasting method for a given workload, one follows the branches by evaluating the identified workload intensity behavior characteristics (cf. Section 7.1.1). The thresholds in the conditions of the branches are parameterized. The values of these thresholds are either derived from the forecasting objectives or based on empirical values obtained during our experiments. The leaves of the decision tree contain one or several recommended forecasting methods. In case a leave contains more than one suitable forecasting method, the forecasting phase executes all of them and evaluates their

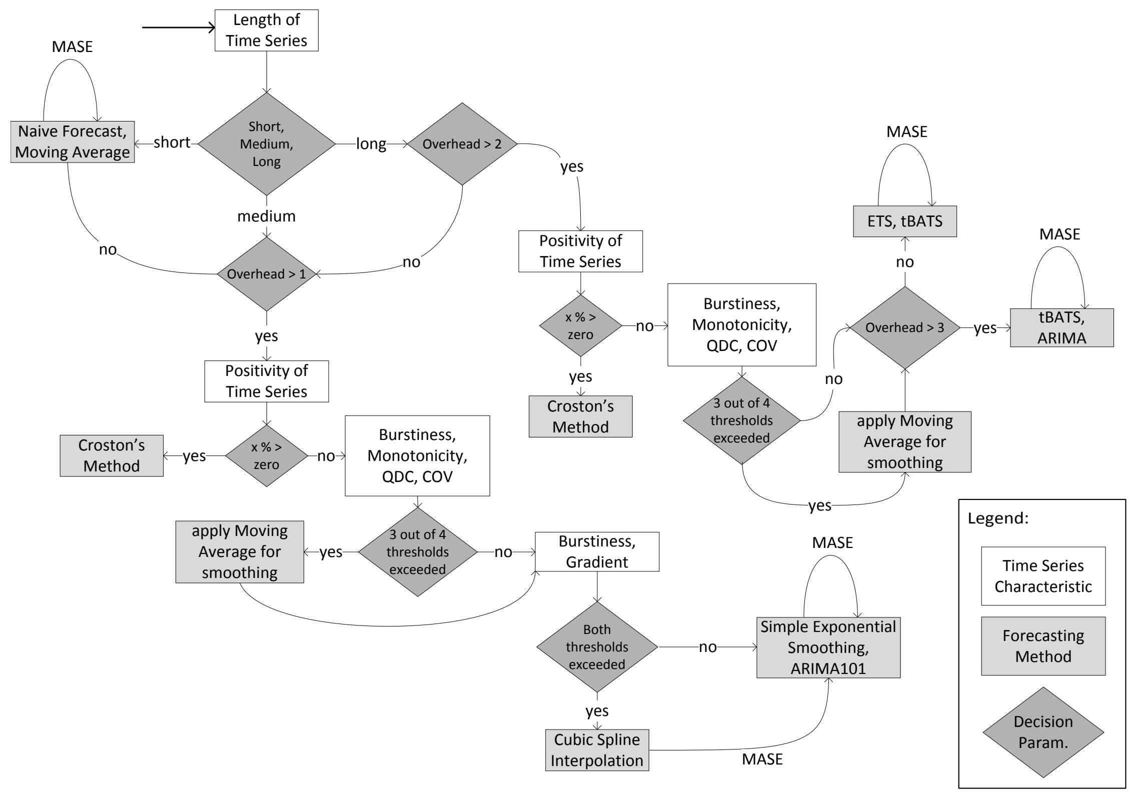

Length of Time Series
MASE
Naive Forecast, Moving Average
short
Short, Medium, Long
long
Overhead > 2
yes
medium
no
Overhead > 1
no
yes
Positivity of Time Series
x % > zero
yes
Croston's Method
no
Burstiness, Monotonicity, QDC, COV
3 out of 4 thresholds exceeded
yes
apply Moving Average for smoothing
no
Burstiness, Gradient
Both thresholds exceeded
no
Simple Exponential Smoothing, ARIMA101
MASE
yes
Cubic Spline Interpolation
MASE
Positivity of Time Series
x % > zero
no
Burstiness, Monotonicity, QDC, COV
yes
Croston's Method
3 out of 4 thresholds exceeded
no
yes
apply Moving Average for smoothing
Overhead > 3
no
ETS, tBATS
MASE
yes
tBATS, ARIMA
MASE
Legend:
Time Series Characteristic
Forecasting Method
Decision Param.

Figure 7.4.: Decision tree for workload classification and selection of appropriate forecasting methods.

estimated forecast accuracy using the MASE metric to select the most accurate result. Before the next iteration of the forecasting phase, i.e., when real system observations are available, we can also determine the accuracy of the forecast values compared to the real observations using the MASE metric. These results are then employed in the decision tree in the next iteration of the classification phase.

7.3. WCF Architecture and Implementation

We implemented our self-adaptive workload classification and forecasting process in Java. In the following, we refer to this implementation as WCF system. For workload forecasting, we used the implementations provided by the forecasting package (Hyndman, 2009) for R, a language and environment for statistical computing (R Project, 2013). Figure 7.5 depicts the architecture of our WCF System as a UML component diagram.

Architecture

The central component of the WCF System is the `WCFManager`. It is responsible for accepting and managing the workload intensity behavior traces to be classified and forecast and the respective configuration settings. The `WCFManager` periodically triggers the classification and forecasting processes according to the forecasting objectives. In addition, it also implements management functionality to persist the given workload intensity behavior traces, configurations, and results. A more detailed description of how to interact with the `WCFSystem` using its interfaces is presented in the following.

The `WIBClassification` component realizes the classification phase of the WCF process and implements the previously presented decision tree. This component receives a

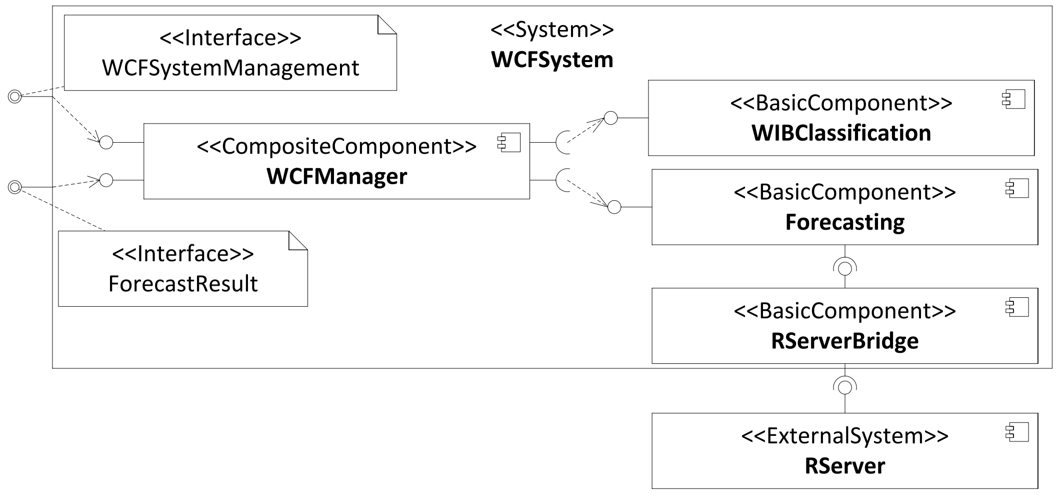


Figure 7.5.: Architecture of the WCF System.

workload intensity behavior trace from the `WCFManager` and selects a forecasting method for this workload intensity behavior according to the decision tree. It returns the result, i.e., the selected WIB class, to the `WCFManager` component.

The `Forecasting` component implements the forecasting phase of our WCF process. It executes the forecasting method that has been selected by the `WIBClassification` component. To obtain forecasting results, the `Forecasting` component uses an external system, the `RServer`. This is basically a TCP/IP server that allows other programs to use the facilities of R, a language and environment for statistical computing (R Project, 2013). To communicate with the `RServer`, the `Forecasting` component uses the `RServer-Bride`. This is a wrapper which encapsulates the controlling of R and that processes the results from the R environment.

Interfaces

The `WCFSystem` provides two interfaces. The `WCFSystemManagement` interface offers management functionality to register new or remove existing workload intensity behaviors, and to read and update the `ForecastingObjectives` of a workload intensity behavior. By setting the forecasting objectives (cf. Section 7.2.1), the user can influence the execution and thus the results of the classification and forecasting processes. In addition, the `WCFSystemManagement` interface can be used to trigger executions of a classification or a forecast manually.

The second interface provided by `WCFSystem` is the `ForecastResult` interface, which is used for data exchange. By default, data is read from and written to buffers, to support the integration of the `WCFSystem` into a pipes-and-filters architecture. The reason for this design decision is that our WCF system should be able to process data provided at tun-time as an input stream by monitoring frameworks. In this case, input data are newly monitored arrival rate values of the registered workload intensity behaviors, which are provided constantly and in fixed but configurable periods. The output data of the `WCFSystem` are forecast results, which are written into a buffer, too. This buffer can then be used by a resource provisioning system or a system adaptation component for further processing. Another option supported by the `WCFSystem` is to write the forecast results to a file for manual result interpretation. More details on the integration of the `WCFSystem` in our model-based adaptation process follows below.

Control Flow

The UML sequence diagram in Figure 7.6 illustrates the control flow of an exemplary use case of the `WCFSystem`. Note that we omitted the `RServerBridge` in this diagram as it simply acts as a wrapper for the interface of the `RServer`.

To register a new workload intensity behavior, a user can invoke the `registerWIB(ts, forecasting_objectives)` method of the `WCFSystemManagement` interface. This triggers the `WCFManager` to create a new record for this workload intensity behavior. The parameters of this method are a reference to the time series buffer as well as the forecasting objectives for this time series. The referred time series buffer in this context is the entity that continuously provides new monitoring data for the time series.

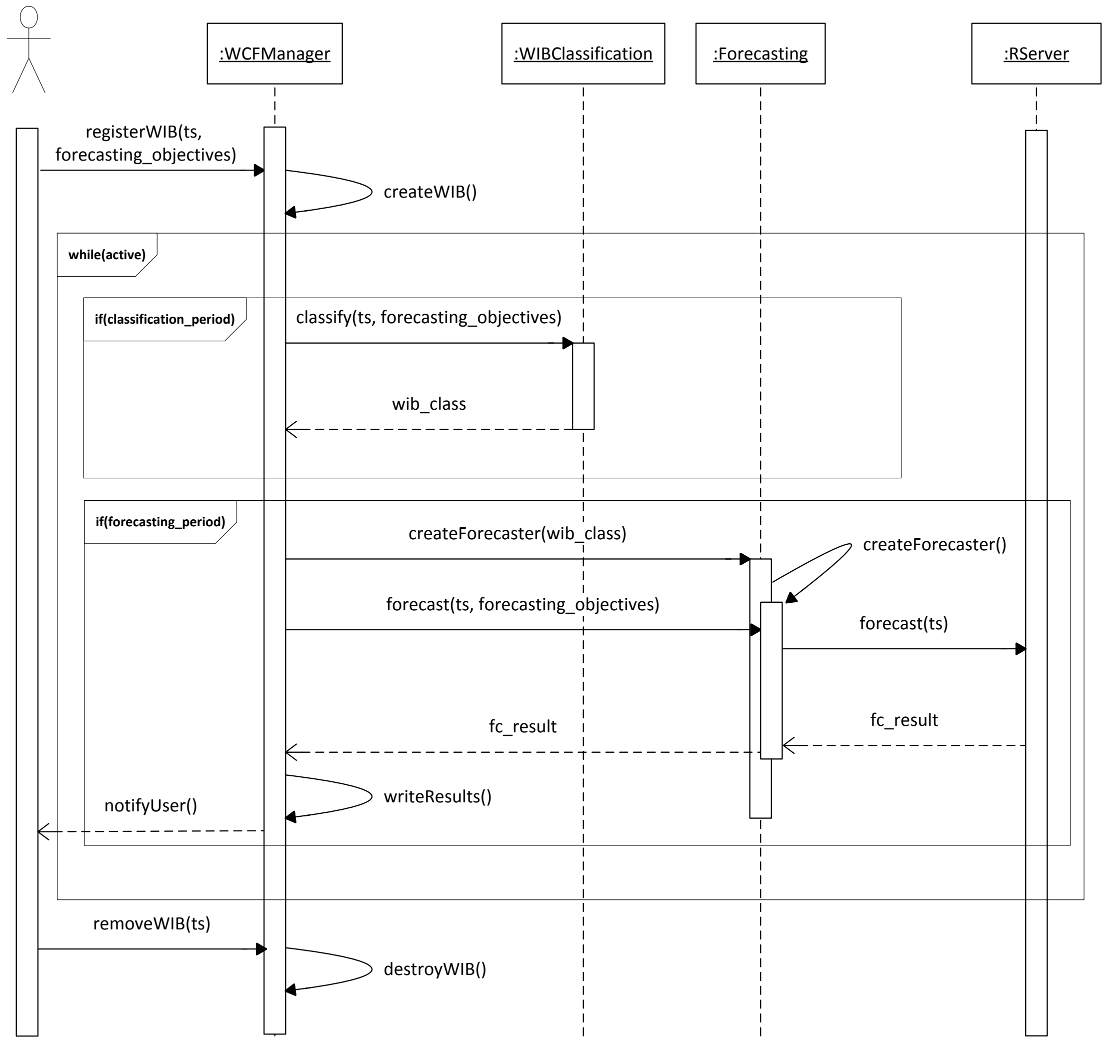


Figure 7.6.: UML sequence diagram illustrating an exemplary use case of the WCF system.

While the `WCFManager` is active, it periodically triggers the classification and forecasting of its registered workload intensity behaviors, depending on the specified time periods. For classification, the `WCFManager` invokes the `classify` method of the `WIBClassification` component. Arguments are a reference to the time series (`ts`) that shall be classified and the specified `forecasting_objectives`. The `WIBClassification` then classifies the workload intensity behavior according to the process we specified in Section 7.2 and returns the result `wib_class` to the `WCFManager` component.

To obtain workload forecasts, the `WCFManager` calls the `createForecaster(wib_class)` method of the `Forecasting` component. This creates a `Forecaster` object which acts as a proxy for the forecasting method specified by the `wib_class` argument and implemented in the `RServer`. Then, the `WCFManager` can pass the time series `ts` and the `forecasting_objectives` to the created `Forecaster` to trigger the forecasting process using the `forecast` method. The `Forecaster` then uses the `RServerBride` to communicate with the `RServer` and start the forecast. When finished, it receives the results from the `RServer`, processes and passes them to the `WCFManager` . The WCFManager stores the results in the configured location and notifies the user that new forecasting results are available.

To remove a workload intensity behavior from the classification and forecasting process, the user can invoke `removeWIB(ts)`. Then, the `WCFManager` component will trigger no further classifications or forecasting.

Integration

An important requirement for the architecture of the WCF system is that it can be integrated in a pipes-and-filters architecture. The reason is that data must be processed at system run-time. According to Buschmann et al. (1996, p. 53), "the pipes-and-filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters."

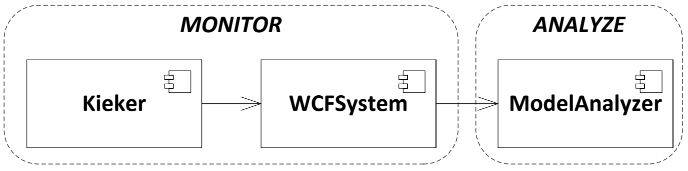


Figure 7.7.: Integration of the `WCFSystem` in the model-based adaptation approach.

Similarly, in our context of model-based system adaptation, the MONITOR phase continuously provides new workload data that must be processed at run-time by our `WCFSystem` and directly provided to the ANALYZE phase. Figure 7.7 sketches a possible integration of the `WCFSystem` into our model-based adaptation control loop (cf. Chapter 4). In this example, Kieker, a framework for monitoring and analyzing a software system's run-time behavior (van Hoorn et al., 2012), continuously delivers monitoring data of the service's request arrival rates. These data are piped into our `WCFSystem` that processes the data at run-time and delivers its forecasts to the `ModelAnalyzer` that can use the forecasts to detect problems for the workload forecasts.

7.4. Evaluation

To evaluate our self-adaptive workload classification and forecasting (WCF) approach, we conduct experiments with different settings and real-world workload intensity behavior traces. The goal of this evaluation is to assess the applicability of our approach and its accuracy compared to other forecasting methods in different situations. For the evaluation, we use two different real-world workload intensity behavior traces. These traces exhibit different characteristics like frequency, trends, seasonal patterns, bursts, noise, etc.

Wikipedia Germany Page Requests: This workload is the time series of the number of page requests per hour at the web-page of Wikipedia Germany. It has been extracted from the publicly available server logs for October, 2011 (Wikimedia Project, 2011).

Transaction Processing This workload contains the transaction arrivals every 15 minutes of a transaction processing service monitored in a mainframe system over one week, from Monday to Sunday. This data has been provided by an industrial collaboration partner.

7.4.1. Experiment Design

In the following four experiments, we assess the accuracy of the forecast results of our self-adaptive WCF approach. We will show that by adapting the forecasting method at run-time to suit the characteristics of the workload, we can improve the forecasting accuracy compared to using a fixed forecasting method. Thus, the basis for our comparison is the accuracy achieved with other forecasting methods that have been selected manually from the list of methods surveyed in Section 7.1. To quantify the forecast result accuracy, we calculate a relative error for every single forecast point.

$$relativeError_t = \frac{|forecastValue_t - observedArrivalRate_t|}{observedArrivalRate_t}$$

In the first experiment, the WCF approach is compared to a fixed use of ETS in a scenario with the transactional workload and no training of the forecasting methods. The experiments II to IV compare a configuration of the WCF approach limited to select only forecasting methods from a certain overhead group to the individual methods within the respective overhead group. In these experiments, we use parts of the workload data as training set for the considered forecasting methods. Additionally, in all experiments we compare the Naive forecasting method to the other forecasting methods. The Naive method is equivalent to using just system monitoring without forecasting, i.e., we can use it as a baseline to quantify and illustrate the benefit of applying forecasting methods in general. In each experiment, the manually selected forecasting methods have been executed with identical forecasting objectives and on the same input data as WCF. This experiment design enables conclusions if the WCF approach successfully classifies the workload intensity behavior or not. Successful classification in this context means that the forecasting method that delivers the highest accuracy for a particular forecast execution is selected by the WCF approach in the majority of cases.

In the following sections, we illustrate the distribution of these relative errors as cumulative distribution functions. Our charts show the inclusive error classes on the x-axis and on the y-axis the corresponding percentage of all forecast points. More formally, a point (x, y) in our charts denotes that y percent of the forecast points have a relative error between 0% and x%. In other words, functions with a trend to the top left corner of the plot are better than functions to the bottom right. In addition, we compute the statistical key indexes (arithmetic mean, median, quartiles, minimum, and maximum) of the error distributions to enable direct comparison. Finally, we use R (R Project, 2013) to conduct directed, paired t-tests to determine if the average forecast accuracy of a certain forecasting method is significantly better than the average forecast accuracy of another method in the context of the respective experiment.

7.4.2. Experiment I: Comparing WCF with ETS and Naive Forecasting

In this experiment, we compare the forecast accuracy of the WCF approach with ETS and the Naive forecasting method. We assume that we have just added a new workload intensity behavior trace to the WCF system, i.e., no historical data is available. The WCF approach can select forecasting methods from all overhead groups. For this experiment, we use the transactional workload intensity behavior over five days, from Monday till

Table 7.2.: Configuration of experiment I.

Experiment Focus	Comparison of WCF to static ETS and Naive
Forecasting Methods	WCF, ETS, Naive
Input Data	Transactional workload, Monday to Friday, 240 values in transactions per 30 minutes, frequency = 48, 5 periods (days)

Horizon
(number of forecast points (h)
for time series length (tsl))
	h = 1 for tsl in [1;24] (1st half period),
	h = 2 for tsl in [25;144] (until 3rd period complete),
	h = 12 for tsl in [145;240] (4th and 5th period)

Friday. The forecast horizon for all methods is configured to increase stepwise, as shown in Table 7.2.

The forecast values of the tested methods and the observation values are plotted in Figure 7.8. In the x-axis, we can see how the WCF approach classifies the given workload. The vertical dashed lines mark the the time point when WCF switches to a higher overhead group. Regarding the forecast values, the chart shows that ETS has several bursts during the first three periods and therefore does not stay as close to the observed values in a number of cases as WCF. Furthermore, during the last forecast executions in the fourth and fifth period (Thursday and Friday), the WCF approach successfully detects the pattern of the days before and therefore estimates the time series better than ETS.

This fact is also reflected in Figure 7.9, showing the cumulative error distribution for each method. The results demonstrate that the WCF approach achieves better forecast accuracy compared to ETS and Naive. Although ETS induces processing overheads of 715 ms per forecast execution, compared to 45 ms for the Naive forecasts (computation of the confidence intervals), it can only partly achieve slightly better forecast accuracy than the Naive method. The WCF approach has an average processing time of 61 ms until the overhead group 4 methods are selected after the first three periods. For the last two periods, the processing time per execution of the WCF approach is 13 seconds on average.

In Table 7.3, we summarize the characteristics of the error distributions of the individual forecasting methods' accuracies using statistical indexes. In addition, the error distributions are illustrated with a box plot in Figure 7.10. The WCF approach shows the lowest median value of 20.7% and the lowest mean value of 47.4%, i.e., on average, WCF is more accurate than the compared forecasting methods. In addition, the WCF approach has a significantly smaller maximum error value, i.e., it is more robust than the compared methods.

Table 7.3.: Experiment I: Characteristics of the error distributions.

Method	**Min.**	**25% Quantile**	**Median**	**Mean**	**75% Quantile**	**Max.**
WCF	0.0128%	9.474%	20.77%	47.39%	49.65%	874.3%
ETS	0.0014%	12.2%	32.31%	75.01%	73.36%	1977%
Naive	0.4917%	16.26%	38.05%	78.88%	81.5%	1671%

In Figure 7.11, the percentage error distributions of WCF and ETS are tested in an paired, directed t-test. The alternative hypothesis is that their true difference in means is less than zero, i.e., there is a significant difference in the mean forecast accuracy of the tested forecasting methods. The result shows that there is a significant mean of differences of

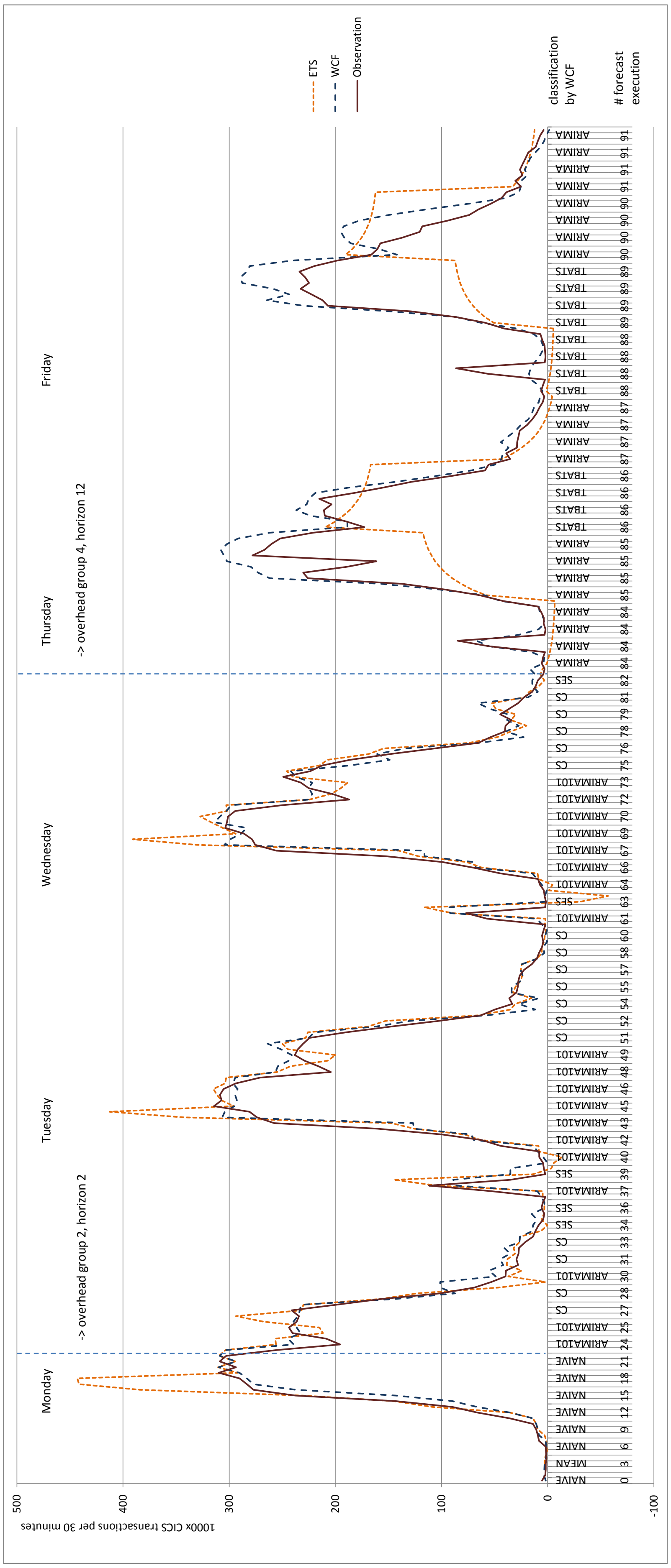
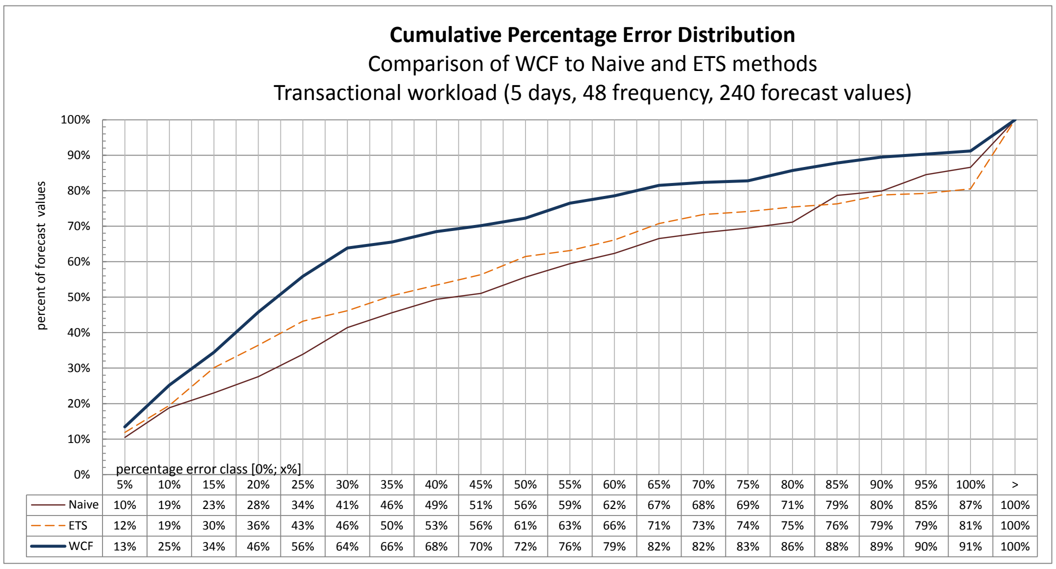


Figure 7.8.: Experiment I: WCF vs. ETS.



	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%	100%	>
Naive	10%	19%	23%	28%	34%	41%	46%	49%	51%	56%	59%	62%	67%	68%	69%	71%	79%	80%	85%	87%	100%
ETS	12%	19%	30%	36%	43%	46%	50%	53%	56%	61%	63%	66%	71%	73%	74%	75%	76%	79%	79%	81%	100%
WCF	13%	25%	34%	46%	56%	64%	66%	68%	70%	72%	76%	79%	82%	82%	83%	86%	88%	89%	90%	91%	100%

Figure 7.9.: Experiment I: Cumulative error distribution of WCF, ETS and Naive forecasts.

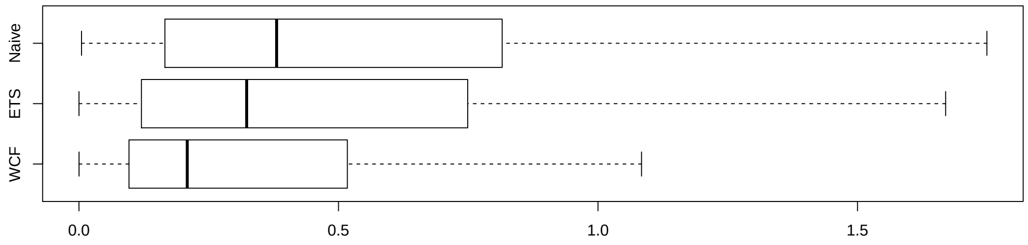


Figure 7.10.: Experiment I: Box plots of the error distributions without outliers.

−0.27. This indicates that the WCF approach achieves a lower forecast error on average than the compared methods. In addition, we tested the percentage error distributions of WCF and the Naive method in the same way. The result of a significant mean of differences of −0.31 confirms that applying the WCF approach is significantly better than simply using system monitoring data (which is equivalent to the Naive method).

7.4.3. Experiment II: Comparing WCF with Forecasting Methods of Overhead Group 2

In this experiment, we configure WCF such that it selects only forecasting methods from overhead group 1 and 2 (WCF2). We then compare this modified WCF2 against the individual forecasting methods contained in overhead group 2 (CS, ARIMA101 and SES). Note that the strength of methods from overhead group 2 is the trend extrapolation. As none of these methods is capable of handling seasonal patterns, high forecast accuracy is not likely to be achieved for the transactional workload intensity behavior that exhibits highly complex daily patterns. To achieve acceptable forecast accuracy, we apply the methods with a high frequency and with a maximum horizon of only two forecast mean values. We use six days of the transactional workload intensity behavior in this experiment from Monday until Saturday. More details of the experiment configuration are given in Table 7.4.

Figure 7.13 depicts the cumulative percentage error distribution for each of the executed methods. This graph shows that WCF achieves a similar forecast accuracy as SES. ARIMA101 and CS show lower forecast accuracy (they are constantly below SES and WCF). As the WCF approach combines the forecasting methods internally through its

```
        Paired t-test

data:  WCF and ETS
t = -2.1129, df = 229,
    p-value = 0.01785
alternative hypothesis:
    true difference in means
    is less than 0
95 percent confidence interval:
    -Inf -0.06031597
sample estimates:
mean of the differences
    -0.2762366
```

```
        Paired t-test

data:  WCF and Naive
t = -2.7639, df = 229,
    p-value = 0.003088
alternative hypothesis:
    true difference in means
    is less than 0
95 percent confidence interval:
    -Inf -0.1267299
sample estimates:
mean of the differences
    -0.3148823
```

Figure 7.11.: Experiment I: Directed, paired t-test on WCF and ETS error distributions (left) and on WCF and Naive error distributions (right).

Table 7.4.: Configuration of experiment II.

Experiment Focus	Comparison of overhead group 2 methods with WCF restricted to select from group 1 and 2
Forecasting Methods	CS, ARIMA101, SES, WCF, Naive
Input Data	Transactional workload, Monday to Saturday, 576 values in transactions per 15 minutes, frequency = 96, 6 periods (days)

Horizon
(number of forecast points (h)
for time series length (tsl))
	h = 1 for tsl in [1;48] (1st half period),
	h = 2 for tsl in [49;576] (until 6th period complete)

classification and feedback mechanism, this experiment shows that the self-adaptation of the WCF approach does not decrease the forecasting accuracy. Regarding the computational overhead, we observe that for each execution all forecasting methods induce approximately the same amount of computational overhead, which is only 55 ms on average. This allows a high frequency of forecast executions.

Figure 7.12 depicts the forecast values of the applied forecasting methods compared to the observation values. This chart illustrates that the WCF approach constantly stays closer to the observed values than ARIMA101 (divergences at the beginning) and CS (which constantly assumes overly strong trends at the edges of seasonal patterns).

Table 7.5.: Experiment II: Characteristics of the error distributions.

Method	**Min.**	**25% Quant.**	**Median**	**Mean**	**75% Quant.**	**Max.**
WCF	0.067%	5.063%	18.64%	59.69%	47.94%	3777%
Naive	0.029%	6.215%	19.26%	53.79%	44%	3779%
SES	0.029%	5.935%	20.13%	54.8%	47.24%	3777%
CS	0.005%	7.232%	21.38%	130.4%	50.57%	6476%
ARIMA101	0.114%	7.223%	25.68%	77.39%	65.5%	3776%

In Table 7.5, the error distributions of the individual forecast strategies are characterized by basic statistical indexes. In addition, the distributions are illustrated as box plots in Figure 7.14. On the one hand, the WCF approach shows the lowest median error of 18.6%. On the other hand, the Naive method achieves the lowest mean error of 53.7%. This shows

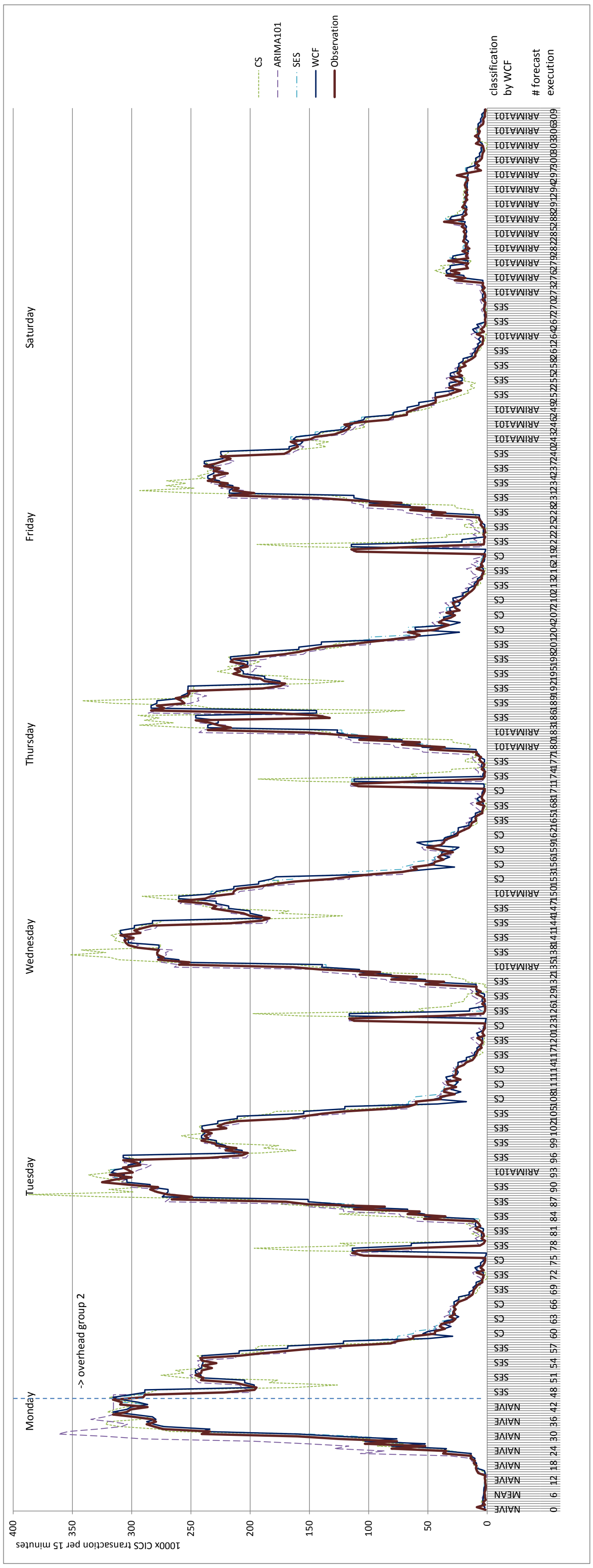
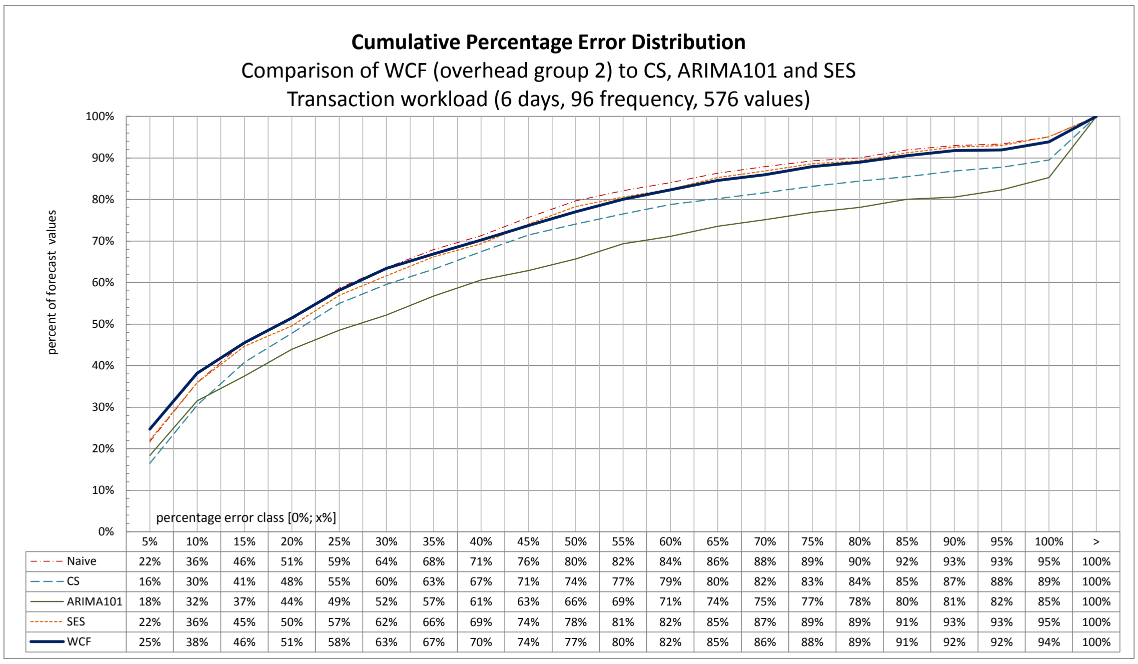


Figure 7.12.: Experiment II: WCF, CS, ARIMA101, and SES.



	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%	100%	>
Naive	22%	36%	46%	51%	59%	64%	68%	71%	76%	80%	82%	84%	86%	88%	89%	90%	92%	93%	93%	95%	100%
CS	16%	30%	41%	48%	55%	60%	63%	67%	71%	74%	77%	79%	80%	82%	83%	84%	85%	87%	88%	89%	100%
ARIMA101	18%	32%	37%	44%	49%	52%	57%	61%	63%	66%	69%	71%	74%	75%	77%	78%	80%	81%	82%	85%	100%
SES	22%	36%	45%	50%	57%	62%	66%	69%	74%	78%	81%	82%	85%	87%	89%	89%	91%	93%	93%	95%	100%
WCF	25%	38%	46%	51%	58%	63%	67%	70%	74%	77%	80%	82%	85%	86%	88%	89%	91%	92%	92%	94%	100%

Figure 7.13.: Experiment II: Cumulative error distribution of WCF, CS, ARIMA101, and SES.

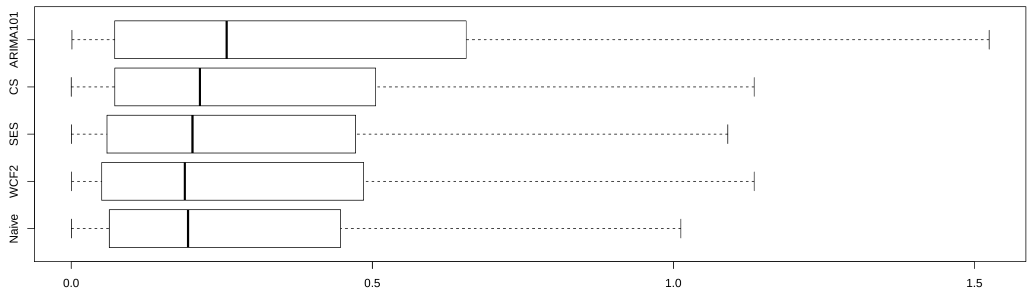


Figure 7.14.: Experiment II: Box plots of the error distributions without outliers.

that in the presence of strong seasonal patterns the simple trend extrapolating forecasting methods are not useful as their accuracy is comparable or worse than the accuracy of the Naive method.

In Figure 7.15, WCF and CS percentage error distributions are tested by a paired, directed t-test on the hypothesis that their true difference in means is less than zero. The result indicates that there is a highly significant mean of differences of -0.70 showing that the WCF approach achieves significantly lower forecast errors. In addition, WCF and ARIMA101 percentage error distributions are tested in the same way with the result of a significant mean of differences of -0.17. These two results underline that the WCF approach achieves higher forecast accuracy by its classification mechanism than the compared methods (CS and ARIMA101).

As the SES method achieved the highest accuracy in this experiment, we expect a high similarity of the percentage error distributions of WCF and SES. Accordingly, we did not detect a significant difference of means in the paired t-test (cf. Figure 7.16). When comparing the error distributions of the WCF approach and the Naive forecasting method, we detected a small but significant mean of differences of 0.058 using the paired, directed t-test. This result could be interpreted as indication that the Naive method is as good as the WCF approach in this scenario. However, this observation would quickly change if the workload intensity behavior stops exhibiting strong seasonal patterns, e.g., when

```
          Paired t-test

data:  WCF and CS
t = -4.8649, df = 570,
   p-value = 7.418e-07
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf -0.4675543
sample estimates:
mean of the differences
   -0.7069751
```

```
          Paired t-test

data:  WCF and ARIMA101
t = -5.7216, df = 570,
   p-value = 8.529e-09
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf -0.1260808
sample estimates:
mean of the differences
   -0.177067
```

Figure 7.15.: Experiment II: Directed, paired t-test on WCF and CS error distributions (left) and on WCF and ARIMA101 error distributions (right).

```
          Paired t-test

data:  WCF2 and SES
t = 1.9554, df = 570,
   p-value = 0.9745
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf 0.09009366
sample estimates:
mean of the differences
   0.04889562
```

```
          Paired t-test

data:  WCF2 and Naive
t = 2.3452, df = 570,
   p-value = 0.00968
alternative hypothesis:
   true difference in means
   is greater than 0
95 percent confidence interval:
   0.0175491       Inf
sample estimates:
mean of the differences
   0.05899171
```

Figure 7.16.: Experiment II: Directed, paired t-test on WCF and SES error distributions (left) and on WCF and Naive error distributions (right).

considering data of higher resolution (seconds, minutes) for short term trend interpolation or considering highly aggregated data for long term trend extrapolation (weeks, months, years). Unfortunately, this could not be validated as no high resolution and long term real-world workload intensity behavior data was available.

7.4.4. Experiment III: Comparing WCF with Forecasting Methods of Overhead Group 3

In this experiment, we compare the forecast accuracy of the WCF approach configured to use only forecasting methods from overhead group 3 (WCF3) with the individual methods contained in overhead group 3 (ETS and tBATS). The strength of these two methods is the seasonal pattern detection. Therefore, at the beginning of the experiment, we train all compared methods using three time periods from the Wikipedia workload. In total, we use 21 days (one week) of the Wikipedia workload intensity trace, three days as training set and the rest for evaluating the forecast accuracy. More details on the experiment configuration are given in Table 7.6.

The cumulative percentage error distribution for each of the executed methods is given in Figure 7.18. The WCF approach and tBATS achieve a similar forecast accuracy. ETS is only slightly worse. The big gap to the Naive forecast indicates that as expected, the complex forecasting methods provide much more accurate results, i.e., they can be beneficial for proactive resource management.

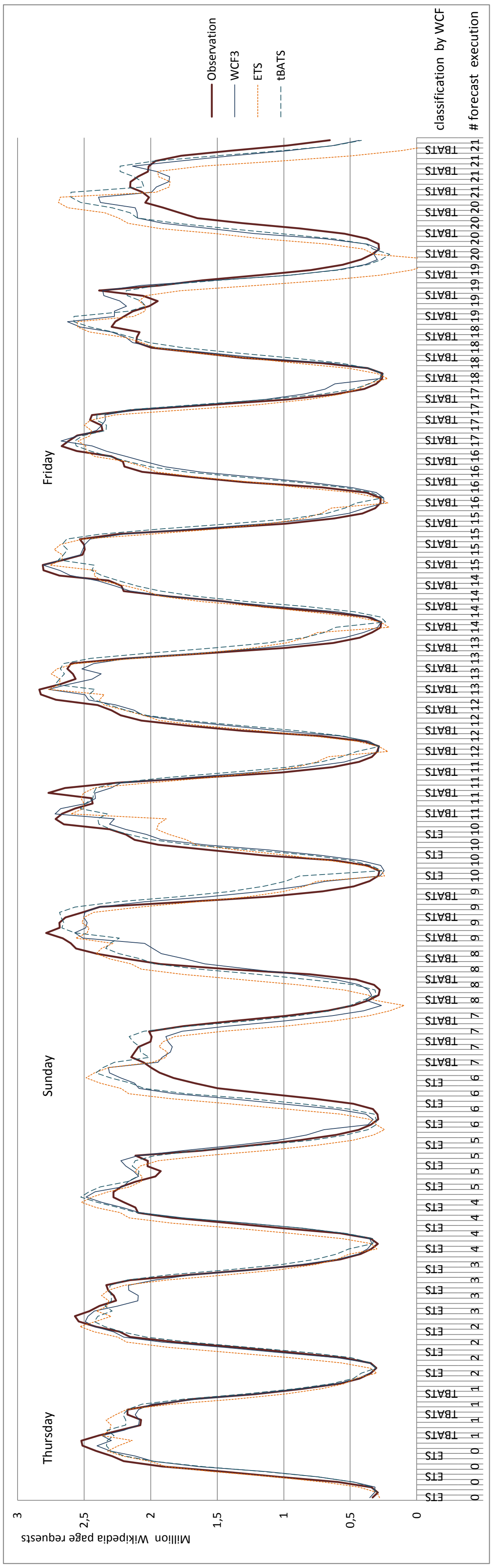
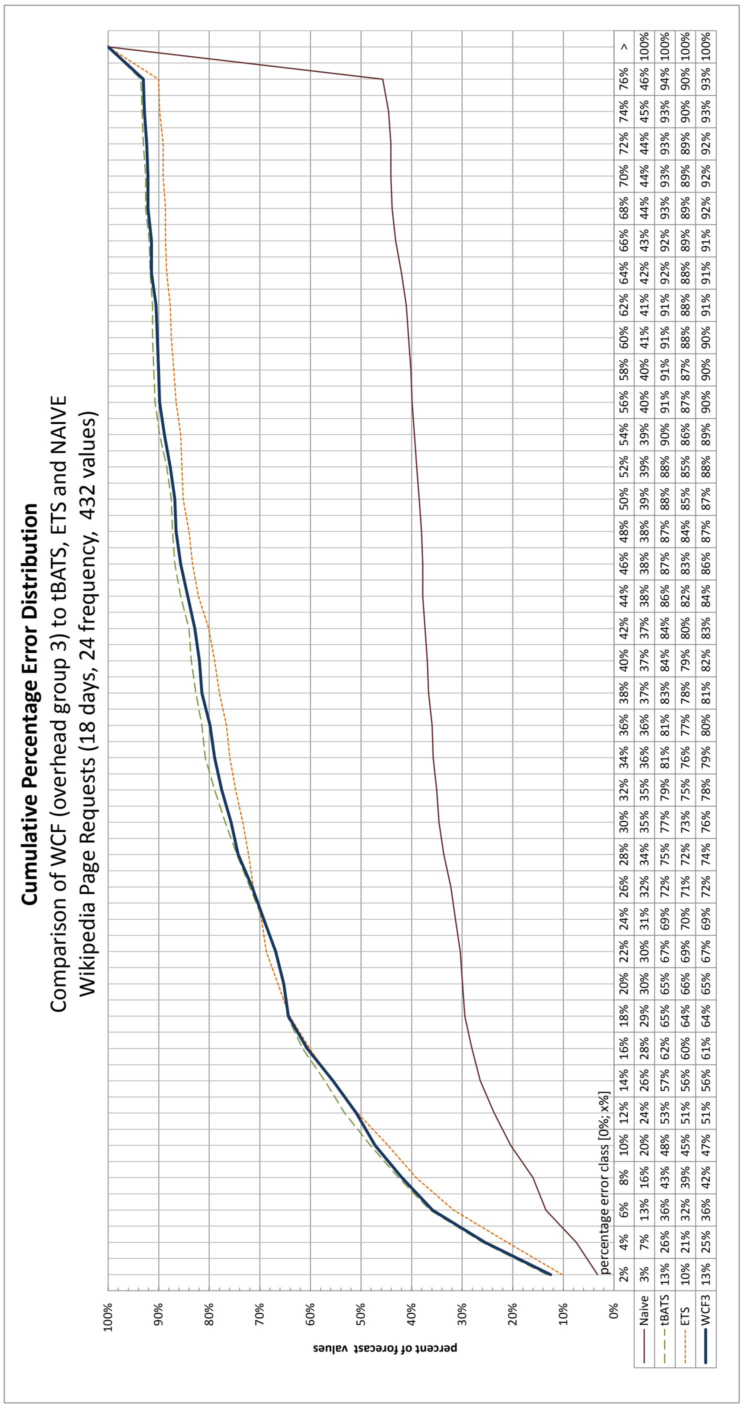


Figure 7.17.: Experiment III: WCF, ETS, tBATS and ETS.



percentage error class [0%; x%]	2%	4%	6%	8%	10%	12%	14%	16%	18%	20%	22%	24%	26%	28%	30%	32%	34%	36%	38%	40%
Naive	3%	7%	13%	16%	20%	24%	26%	28%	29%	30%	30%	31%	32%	34%	35%	35%	36%	36%	37%	37%
tBATS	13%	26%	36%	43%	48%	53%	57%	62%	65%	65%	67%	69%	72%	75%	77%	79%	81%	81%	83%	84%
ETS	10%	21%	32%	39%	45%	51%	56%	60%	64%	66%	69%	70%	71%	72%	73%	75%	76%	77%	78%	79%
WCF3	13%	25%	36%	42%	47%	51%	56%	61%	64%	65%	67%	69%	72%	74%	76%	78%	79%	80%	81%	82%

percentage error class [0%; x%]	42%	44%	46%	48%	50%	52%	54%	56%	58%	60%	62%	64%	66%	68%	70%	72%	74%	76%	>
Naive	37%	38%	38%	38%	39%	39%	39%	40%	40%	41%	41%	42%	43%	44%	44%	44%	45%	46%	100%
tBATS	84%	86%	87%	87%	88%	88%	90%	91%	91%	91%	91%	92%	92%	93%	93%	93%	93%	94%	100%
ETS	80%	82%	83%	84%	85%	85%	86%	87%	87%	88%	88%	88%	89%	89%	89%	89%	90%	90%	100%
WCF3	83%	84%	86%	87%	87%	88%	89%	90%	90%	90%	91%	91%	91%	92%	92%	92%	93%	93%	100%

Figure 7.18.: Experiment III: Cumulative error distribution of WCF, ETS, tBATS and Naive.

Table 7.6.: Configuration of experiment III.

Experiment Focus	Comparison of overhead group 3 forecasting methods with WCF that is restricted to select from group 3
Forecasting Method	ETS, tBATS, WCF, Naive
Input Data	Wikipedia workload, 3 weeks, 504 values in page requests per hour, frequency = 24, 21 periods as days, training set = first 3 days (Monday to Wednesday)
Horizon (number of forecast points (h) for time series length (tsl))	h = 12 for tsl in [73;504] (4^{th} until 21^{st} period), no forecasts for the first 3 periods

All three forecast strategies induce approximately the same amount of computational overhead as they belong to the same overhead group. On average, tBATS needs 10 seconds per execution, ETS 14 seconds and WCF 19 seconds (as it executes both tBATS and ETS every second time). In this experiment, their computations are based on the last seven observed periods, i.e., on a maximum number of 168 time series points.

The forecast values of the individual methods and the observation values for the first ten days of this experiment are plotted in Figure 7.17. The chart illustrates the high forecast accuracy of all three forecasting methods, especially at the edges of the daily seasonal patterns. It also shows that the changes in the amplitudes of the daily patterns, e.g., from Sunday to Monday, induce higher forecast errors than the rather constant amplitudes of working days.

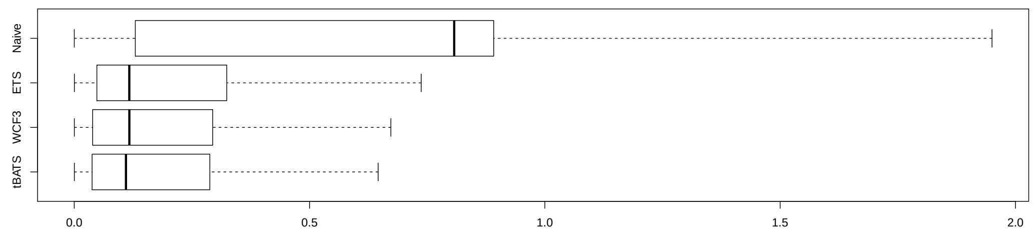


Figure 7.19.: Experiment III: Box plots of the error distributions without outliers.

Table 7.7.: Characteristics of the error distributions of experiment III.

Method	Min.	25% Quant.	Median	Mean	75% Quant.	Max.
tBATS	0.0209%	3.793%	10.93%	22.76%	28.33%	185.2%
WCF	0.0209%	3.915%	11.63%	24.01%	29.42%	209.6%
ETS	0.0294%	4.807%	11.62%	27.44%	32.41%	241.2%
Naive	0.0081%	12.98%	80.73%	127.4%	89.13%	1013%

Table 7.7 summarizes the characterizing statistical indexes of the error distributions of the compared forecasting methods. In addition, box plots are depicted in Figure 7.19. In this experiment, tBATS shows the lowest median error value of only 10.9%. The low maximum errors of all three forecasting methods underline the benefit of the forecast values obtained by applying complex methods compared to using the Naive method. The experiment also shows that, as the WCF approach is a combination of tBATS and ETS, it cannot be better than these for individual executions. However, the error values are more close to the tBATS method which performs best in this scenario.

When comparing the error distributions of the WCF approach and ETS using a paired,

```
        Paired t-test

data:  WCF and ETS
t = -4.1869, df = 430,
   p-value = 1.716e-05
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf -0.02077467
sample estimates:
mean of the differences
   -0.03426498
```

```
        Paired t-test

data:  WCF and tBATS
t = 1.1914, df = 430,
   p-value = 0.2341
alternative hypothesis:
   true difference in means
   is not equal to 0
95 percent confidence interval:
   -0.008086287  0.032979603
sample estimates:
mean of the differences
   0.01244666
```

Figure 7.20.: Experiment III: Directed, paired t-test on WCF and ETS error distributions (left) and on WCF and tBATS error distributions (right).

directed t-test (cf. Figure 7.20), the computed mean of differences is small but significant having a value of -0.034. This means that the WCF approach constantly achieves a slightly higher accuracy. The mean of differences between WCF's and tBATS' error distributions is insignificant. The reason is that in this experiment, the WCF approach selected tBATS as forecasting method in about 80% of the forecast executions, which is good, as it demonstrates that WCF is able to continuously select the better performing forecasting method.

7.4.5. Experiment IV: Comparing WCF with Forecasting Methods of Overhead Group 4

In this experiment, we compare the forecast accuracy of the WCF approach using only overhead group 4 forecasting methods (WCF4) with the individual forecasting methods of overhead group 4 (ARIMA and tBATS) and the Naive forecasting method. The configuration of this experiment is the same as for experiment IV and is summarized in Table 7.8.

Table 7.8.: Configuration of experiment IV.

Experiment Focus	Comparison of overhead group 4 forecasting methods with WCF restricted to select from group 4
Forecast Strategies (overhead group)	ARIMA, tBATS, WCF4, Naive
Input Data	Wikipedia workload, 3 weeks, 504 values in page requests per hour, frequency = 24, 21 periods as days, training set = first 3 days (Monday to Wednesday)
Horizon (number of forecast points (h) for time series length (tsl))	h = 12 for tsl in [73;504] (4^{th} until 21^{st} period), no forecasts for the first 3 periods

The cumulative percentage error distribution for each of the executed forecasting methods is shown in Figure 7.22. ARIMA and the WCF approach achieve both a similar forecast accuracy with tBATS being only slightly worse. Like in experiment III, the big gap to the Naive forecasting method shows that the forecast values obtained by applying such complex forecasting methods can be beneficial for proactive resource management. In

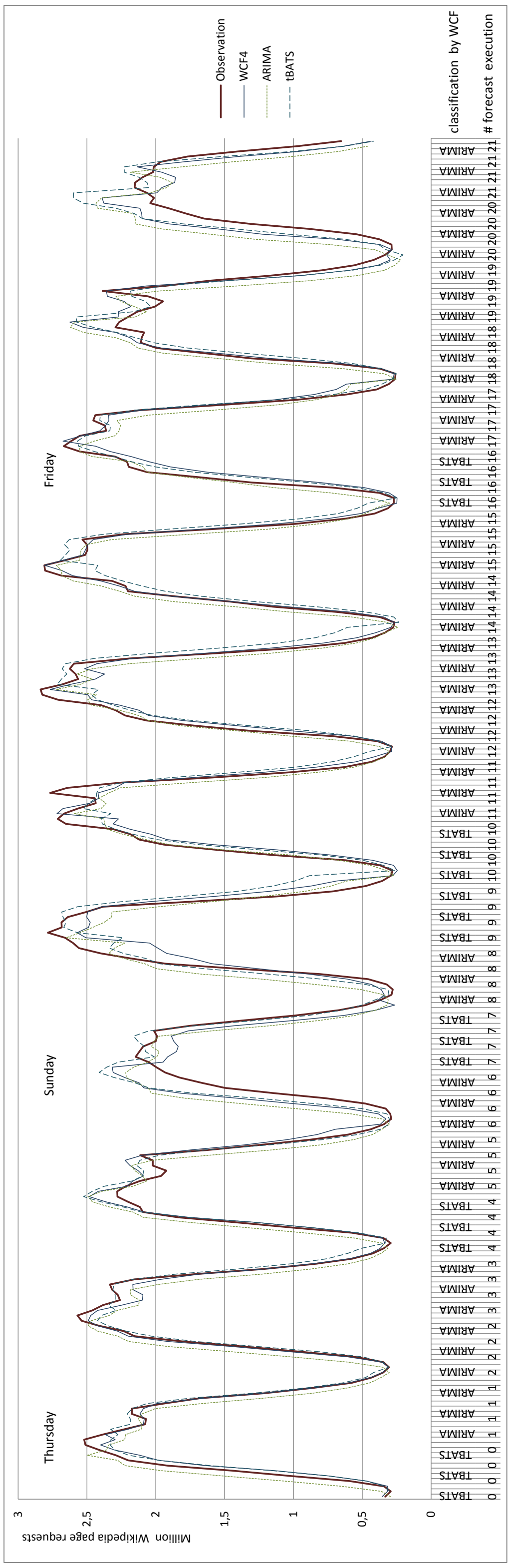
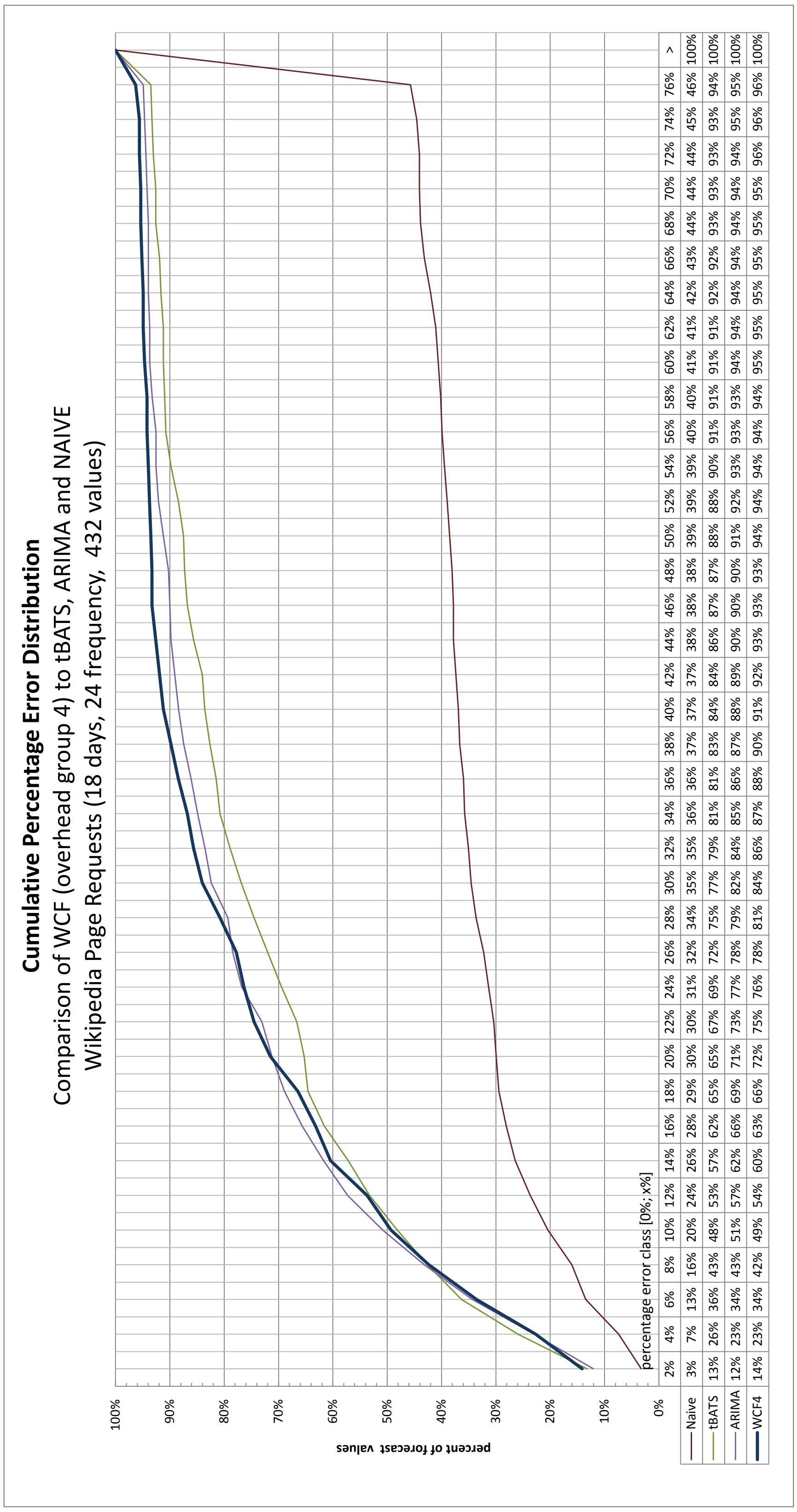


Figure 7.21.: Experiment IV: Comparison of WCF, tBATS, ARIMA.



	2%	4%	6%	8%	10%	12%	14%	16%	18%	20%	22%	24%	26%	28%	30%	32%	34%	36%	38%	40%	42%	44%	46%	48%	50%	52%	54%	56%	58%	60%	62%	64%	66%	68%	70%	72%	74%	76%	>
Naive	3%	7%	13%	16%	20%	24%	26%	28%	29%	30%	30%	31%	32%	34%	35%	35%	36%	36%	37%	37%	37%	38%	38%	38%	39%	39%	39%	40%	40%	41%	41%	42%	43%	44%	44%	44%	45%	46%	100%
tBATS	13%	26%	36%	43%	48%	53%	57%	62%	65%	65%	67%	69%	72%	75%	77%	79%	81%	81%	83%	84%	84%	86%	87%	87%	88%	88%	90%	91%	91%	91%	91%	92%	92%	93%	93%	93%	93%	94%	100%
ARIMA	12%	23%	34%	43%	51%	57%	62%	66%	69%	71%	73%	77%	78%	79%	82%	84%	85%	86%	87%	88%	89%	90%	90%	90%	91%	92%	93%	93%	93%	94%	94%	94%	94%	94%	94%	94%	95%	95%	100%
WCF4	14%	23%	34%	42%	49%	54%	60%	63%	66%	72%	75%	76%	78%	81%	84%	86%	87%	88%	90%	91%	92%	93%	93%	93%	94%	94%	94%	94%	94%	95%	95%	95%	95%	95%	95%	96%	96%	96%	100%

Figure 7.22.: Experiment IV: Cumulative error distribution of WCF, tBATS, ARIMA, and Naive.

addition, the results illustrate the even higher forecast accuracy of the WCF4 approach and the ARIMA method compared to the WCF3 approach in experiment III.

The computational overhead per forecast execution is on average 22 seconds for the WCF approach and ten seconds for tBATS. ARIMA required 15 seconds on average per forecast execution but the measurements showed a higher variance with a maximal duration of 56 seconds. The computations of the forecasts are based on the last seven observed periods, i.e., on a maximum number of 168 time series values.

The forecast values of the individual forecasting methods and the actual observation values are plotted in Figure 7.21 for the first ten days of the workload. The shapes of the time series in this experiment show similar characteristics as in experiment III with even closer estimations of the pattern amplitudes.

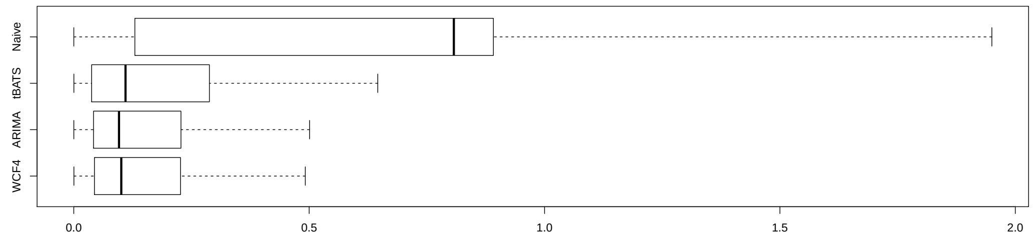


Figure 7.23.: Experiment IV: Box plots of the error distributions without outliers.

Table 7.9.: Characteristics of the error distributions of experiment IV.

Method	**Min.**	**25% Quant.**	**Median**	**Mean**	**75% Quant.**	**Max.**
WCF	0.0272%	4.389%	10.12%	17.49%	22.67%	125.5%
ARIMA	0.0096%	4.193%	9.608%	19.89%	22.77%	340.6%
tBATS	0.0209%	3.793%	10.93%	22.76%	28.33%	185.2%
Naive	0.0081%	12.98%	80.73%	127.4%	89.13%	1013%

In Table 7.9, we summarize the characterizing statistical indexes of the error distributions of the compared forecasting methods. In addition, Figure 7.23 depicts the box plots of the error distributions. ARIMA shows the lowest median value of only 9.6%. The WCF4 approach has the lowest mean error values and achieves this improvement by choosing the better performing method. In this scenario, both methods (ARIMA and tBATS) are selected with a similar probability resulting in a combination of their strengths for particular situations. In addition, WCF4 has the lowest maximum error and therefore the highest trustworthiness from the compared methods. These results show that a measurable improvement can be achieved by using our self-adaptive WCF approach.

The comparison of the WCF approach with ARIMA and tBATS in paired, directed t-tests detects a highly significant mean of differences within the percentage error distributions in both cases (cf. Figure 7.24). This indicates that the WCF approach is able to correctly select the forecasting method that is more likely to show higher accuracy. This demonstrates that our approach of dynamically selecting forecasting methods at run-time has no negative impact on the forecasting accuracy.

If we use the directed, paired t-test to compare the error distributions of WCF4 and WCF3 from experiment III (cf. Figure 7.25), a highly significant mean of differences of -0.65 is observed. The WCF4 approaches' internal use of ARIMA has obviously caused this improvement in accuracy of WCF4 compared to WCF3. The mean of differences of -1.1 is detected when comparing the WCF4 approach to the Naive method using the directed, paired t-test. This large mean of differences of the percentage errors indicates the high potential of the WCF4 approach in this scenario to deliver accurate forecast results.

```
        Paired t-test

data:  WCF4 and ARIMA
t = -1.843, df = 430,
   p-value = 0.03301
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf -0.002530368
sample estimates:
mean of the differences
   -0.02396612
```

```
        Paired t-test

data:  WCF4 and tBATS
t = -4.2109, df = 430,
   p-value = 1.55e-05
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf -0.03208695
sample estimates:
mean of the differences
   -0.05272829
```

Figure 7.24.: Directed, paired t-test on WCF and ARIMA error distributions (left) and on WCF and tBATS error distributions (right).

```
   Paired t-test

data:  WCF4 and WCF3
t = -5.255, df = 430,
   p-value = 1.168e-07
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf -0.04473058
sample estimates:
mean of the differences
   -0.06517494
```

```
        Paired t-test

data:  WCF4 and Naive
t = -11.6577, df = 430,
   p-value < 2.2e-16
alternative hypothesis:
   true difference in means
   is less than 0
95 percent confidence interval:
   -Inf -0.9436938
sample estimates:
mean of the differences
   -1.099108
```

Figure 7.25.: Experiment IV: Directed, paired t-test on WCF4 and WCF3 error distributions (left) and on WCF4 and Naive error distributions (right).

7.5. Summary

In this chapter, we presented an approach for self-adaptive workload classification and forecasting (WCF) at run-time. The approach automatically identifies relevant characteristics of the considered workload intensity behavior and selects suitable forecasting methods according to the configured user-specific forecasting objectives. Especially at the beginning of an observed workload intensity series, when limited historical data is available, a static decision made by a user may not be appropriate for the whole lifetime of the workload intensity. In such cases, the dynamic design of our approach and the flexibility to react on changes in the workload intensity behavior enables WCF to continuously adapt its classifications based on the observed forecasting accuracy, thereby increasing the overall accuracy of the forecast results. Our approach supports online and continuous forecast processing with controllable computational overheads, realized by scheduling workload intensity behavior classifications and forecasting method executions in configurable periods.

We evaluated the applicability and the forecasting accuracy of the proposed approach in four experiments with different settings and workloads. In all experiments, the processing times of all forecasting methods remained within the boundaries specified by their corresponding overhead groups in such a way that the forecast results were available before their corresponding request arrival rates could be observed through monitoring. Moreover, the results demonstrated that our approach can significantly improve the forecasting accuracy with an acceptable additional computational overhead. Thus, we conclude that our approach can be effectively applied at run-time to provide workload forecasts for proactive performance and resource management.

Part III.

Validation and Conclusion

8. Validation

In this chapter, we evaluate how the individual contributions that have been presented and evaluated in isolation in the Chapters 5 to 7 can be integrated into a holistic model-based approach for autonomic performance-aware resource management. In Section 5.4, we evaluated the capability of the modeling abstractions of the resource landscape meta-model to describe the performance relevant aspects of the infrastructure of modern IT systems and services. In Section 6.4, we assessed the applicability of the adaptation points and adaptation process meta-models to describe dynamic system aspects and adaptation processes. Furthermore, an evaluation of the performance prediction capabilities of the modeling abstractions and their suitability for online performance prediction has been presented as part of the work of (Brosig, 2014). What is currently missing is an end-to-end validation of our approach, covering all of its constituent parts. In this chapter, we close this gap and present the end-to-end evaluation and validation of our approach in the context of three representative and real-life case studies, demonstrating the benefits of autonomic performance and resource management in modern dynamic IT systems, infrastructures and services.

This chapter is structured as follows. First, in Section 8.1, we discuss the goals of our validation and formulate specific questions to be answered as part of the validation. Next, we present three detailed case studies addressing the specific validation goals and questions. The first case study, presented in Section 8.2, is based on the SPECjEnterprise2010 benchmark deployed in a virtualized cluster environment. In this case study, we add/remove virtual CPUs, start/stop VMs, or migrate VMs in response to different changes in the system environment (e.g., workload changes, new services that are composed or deployed at run-time) to maintain the performance and resource efficiency of the system. The second case study, presented in Section 8.3, evaluates the applicability of our Workload Classification and Forecasting (WCF) approach for proactive system adaptation at run-time. In this case study, we use realistic workloads to investigate the potential of exploiting workload forecasts for proactive system adaptation. In the third case study, presented in Section 8.4, we apply our approach in the context of our industrial partner Blue Yonder. In this case study, we evaluate the applicability of our approach in a heterogeneous resource environment, assessing its capability to trade-off different performance requirements of multiple customers while maintaining resource efficiency. Finally, in Section 8.5, we summarize the results and discuss their external validity.

8.1. Validation Goals

In Chapter 1, we claim to provide a holistic model-based approach for autonomic performance-aware resource management. We summarize the goal of our validation in the following sentence:

> *The concepts and modeling abstractions presented in this thesis are suitable to model proactive or reactive system adaptation processes; architecture-level performance models can be used to evaluate at run-time the impact of possible adaptation actions for adaptation decisions to maintain multiple performance requirements and resource efficiency.*

We structure the validation goal into three major parts: the evaluation of the capabilities of the developed modeling abstractions with respect to describing the performance-relevant behavior and the adaptation processes of systems, the evaluation of the prediction capabilities of the architecture-level performance model for automated system adaptation, and the end-to-end evaluation of the integration of these concepts to realize the proactive model-based adaptation approach presented in this thesis. In the following sections, we discuss these parts of the validation in more detail.

8.1.1. Modeling Capabilities

When developing new modeling abstractions, it is important to evaluate their expressiveness and appropriateness to meet the intended purpose. The evaluation of these properties for the modeling abstractions developed in this thesis, the resource landscape meta-model and the adaptation points and adaptation process meta-models, has already been presented in Sections 5.4 and 6.4. In Section 5.4, we showed how to use the resource landscape meta-model to model modern distributed data centers and illustrated the advantages of the developed modeling abstractions in a VM (re-)deployment scenario. The results demonstrated that modeling the resource landscape with its hierarchy provides important information that can be used to support adaptation decisions, e.g., to exclude migration targets or to find the most suitable target. In Section 6.4, we evaluated the flexibility and suitability of the adaptation points and adaptation process meta-model to specify system adaptation processes by comparing them with an existing classification of adaptation models (Vogel and Giese, 2012). In addition, we demonstrated the potential of the modeled adaptation processes for autonomic system adaptation by comparing it with optimization heuristics implemented in PerOpteryx (Martens et al., 2010), a framework for automated software architecture improvement. The results showed that our adaptation language and adaptation framework fulfill all essential requirements for adaptation models and demonstrated how the adaptation process meta-model can guide the system adaptation. In summary, the results showed that the modeling formalisms presented as part of this thesis are sufficiently expressive to provide the information needed to support run-time system adaptation and that they can be effectively used to describe system adaptation processes. Note that the evaluation of the expressiveness of the application architecture meta-model is part of the work of Brosig (2014).

8.1.2. Prediction Capabilities of the Architecture-Level Performance Model

An essential element for the success of the approach presented in this thesis is the usability of the employed online performance prediction techniques for performance and resource management at run-time. Thus, it is important to evaluate if the architecture-level performance model as part of DML enables performance predictions that are sufficiently accurate to support adaptation decisions and to drive autonomic decision-making. Given that the development of the online performance prediction techniques is part of the work of Brosig, the validation of these techniques can be found in Brosig (2014). Nevertheless, we include details about the architecture-level performance models that we created for each case study as well as the obtained prediction accuracy in the respective sections. The results show that the provided modeling abstraction levels are suitable to tailor the prediction process to given accuracy and run-time requirements and that the results of the online performance prediction can be effectively used for run-time system adaptation.

8.1.3. End-to-End Validation of the Model-Based Adaptation Approach

The third essential part of the validation is the evaluation of all constituent parts of our approach as a whole. It is important to show that the integration is capable of enabling autonomic performance-aware resource management and to analyze how the approach performs in different situations and under varying conditions. In the following, we discuss the goals of this part of the validation.

In contrast to the evaluation presented in the respective sections of Chapters 5 to 7 considering the individual parts of our approach in isolation, the purpose of the validation presented in this chapter is to validate our approach end-to-end. Our preferred validation method would be to use a benchmark or some kind of widely accepted exemplary scenario to evaluate the quality and performance of our approach and to compare it against other approaches. However, creating a benchmark for self-adaptive software systems is still a challenge as no practical way to characterize self-adaptation capabilities currently exists, especially when comparing alternative systems concerning performance and dependability (Almeida and Vieira, 2011). To give an example, it is a challenge to evaluate not only the impact of an adaptation on the performance and dependability of the system, but also the overall impact of the adaptation with respect to subsequent changes. Thus, we decided to set up our own case studies for assessing and analyzing the behavior of our approach in different situations and under varying conditions. The case studies were carefully selected to ensure the external validity of the results. The systems considered in the case studies are as realistic as possible without making any system-specific assumptions, and the selected evaluation scenarios are based on real-life problems, e.g., of our industrial partner Blue Yonder. To create realistic setups for our experiments, we employed technologies that are extensively used in industry and studied applications from different domains (business information systems and compute-intensive applications). Finally, we also used different real-life workload traces from the Wikimedia Project (2011) as well as traces provided by our industrial collaboration partners. Briefly, the case studies we use for our validation and their respective goals are:

Case Study 1: Here we apply our model-based approach in a homogeneous and virtualized resource environment to allocate resources dynamically in order to accommodate changes in the environment, such as varying workload intensities or deployment of new services. As target application, we use the SPECjEnterprise2010 benchmark (Section 8.2). The goal of this case study is to demonstrate how we can take advantage of the possibilities provided by virtualization to build a model-based adaptation process that adapts the system at run-time to changes in the application workloads or in the system configuration.

Case Study 2: Here we apply our Workload Classification and Forecasting (WCF) approach on real-life workload traces showing how it can be used to proactively adapt systems to changes in their workloads (Section 8.3). The goal of this case study is to demonstrate how WCF can be used to reduce SLA violations through proactive resource provisioning at run-time.

Case Study 3: Here we apply our model-based system adaptation approach in the context of an example project of our industrial partner Blue Yonder to evaluate how it performs in heterogeneous resource environments with multiple different performance requirements (Section 8.4). Compared to case study 1, the goal of this case study is to evaluate whether our approach is applicable in an environment with heterogeneous resources and whether it can be used to trade-off different performance requirements of multiple customers.

With these case studies, our goal is to demonstrate that we are able to leverage WCF and online performance prediction to realize the holistic model-based system adaptation process presented in Chapter 4. The case studies are intended to help us answer several specific questions presented in the following.

Q1: Can our proposed architecture-level model-based approach be effectively used to enable autonomic performance-aware resource management?

To answer this question, we apply our approach in the context of the described case studies to evaluate whetherit is capable of improving the resource efficiency of the studied systems under varying conditions. In these case studies, we model the system performance behavior using the modeling formalism presented in Chapter 5 as well as the system's adaptation process that maintains its operational goals, specified with the modeling formalism presented in Chapter 6. We apply the adaptation framework presented in Section 6.3 to interpret and execute the modeled adaptation process. Thereby, we can assess if the presented online performance prediction and model-based system adaptation techniques can be combined to maintain the system performance and resource efficiency requirements.

Furthermore, we claim that our model-based approach can be effectively used for *proactive* resource provisioning. Thus, we have to validate whether the workload intensity forecasts of the WCF approach presented in Chapter 7 can be used to adapt the system proactively. This leads to the question:

Q2: Can the workload classification and forecasting mechanism be effectively used to enable proactive resource provisioning?

This question is addressed in Section 8.3, where we apply WCF to realistic workloads to predict future workload intensities. We use the workload forecasts as input to our model-based system adaptation approach in order to evaluate if the presented techniques can be used for proactive system adaptation and if they provide benefits compared to trigger-based approaches. To answer if the workload forecasts can be used for proactive system adaptation, we analyze the effect of our proactive adaptation approach on metrics like resource efficiency and SLA violations.

A further important question to investigate is the effectiveness of our approach compared to other approaches.

Q3: How does our approach perform compared to other approaches?

As mentioned previously and discussed by Almeida and Vieira (2011), it is currently a challenging task to benchmark self-adaptive software systems and thereby compare related approaches. Thus, to assess the effectiveness of our approach, we compare it against a static resource allocation approach as well as against a reactive, trigger-based approach. Such approaches are common practice in industry and thus serve as a baseline. To quantify the difference of our approach to the other approaches, we directly compare metrics like SLA violations and resource efficiency. Also, when comparing the different approaches, it is important to consider other metrics such as the computational overhead or the effect and frequency of adaptation actions.

Another claim of this thesis is that our approach is capable of modeling adaptation processes that trade-off different adaptation goals, e.g., different performance requirements of diverse customers. This leads us to the question:

> *Q4: Does our approach support adaptation processes for trading-off multiple objectives?*

This question is addressed in the case study presented in Section 8.4 which considers multiple customers with diverging performance requirements. In the scenarios of this case study, we analyze how the modeled adaptation process adjusts the assigned resources to changes in the customer requirements, considering the performance requirements of other customers sharing the resources.

In each of the following sections, we first discuss the individual goals of the respective case study with respect to the formulated validation questions. We then present in detail how our approach is applied to the specific case study and discuss the results of the conducted experiments. Finally, Section 8.5 gives a summary of the results of all case studies and discusses in detail the answers to the presented questions as well as the validity of the results of our end-to-end validation.

8.2. Model-Based Resource Allocation in Virtualized Environments

In this case study, we apply our approach in a virtualized cluster environment. As application, we selected the SPECjEnterprise2010[1] benchmark which is designed to represent a state-of-the-art enterprise system. The goal of the case study is to demonstrate how we can build a model-based adaptation process that exploits the flexibility of virtualization to adapt the system at run-time to changes in application workloads (Validation Question Q1). Furthermore, to answer validation question Q3, we evaluate the efficiency of our approach compared to static resource assignment.

The presentation of this case study is structured as follows. First, we give an overview of the SPECjEnterprise2010 benchmark and the adaptation process that we use in this case study (Section 8.2.1). Then, in Section 8.2.2, we present the employed architecture-level performance model of the application and the modeled adaptation process. Finally, we present the evaluation results in Section 8.2.3.

8.2.1. Overview of the SPECjEnterprise2010 Benchmark and the Employed Adaptation Process

In this section, we explain the architecture of the SPECjEnterprise2010 benchmark, the hardware infrastructure, and the deployment of the software components that we use in our experiments. Furthermore, we introduce the system's degrees of freedom that we leverage for system adaptation. Based on these degrees of freedom, we specify a generic resource allocation algorithm to maintain the performance requirements and resource efficiency of the system.

[1]SPECjEnterprise2010 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 results or findings in this thesis 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`.

8.2.1.1. Architecture of the SPECjEnterprise2010 Benchmark

SPECjEnterprise2010 is a standard benchmark developed by SPEC's Java subcommittee providing a representative workload to evaluate the end-to-end performance and scalability of Java EE-based application servers. The benchmark workload is generated by an application that is modeled after an automobile manufacturer, comprising 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 (cf. Figure 8.1). 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.

The application logic in the three domains is implemented using Enterprise JavaBeans (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 is 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., via Remote Method Invocation (RMI). As shown in Figure 8.1, the system under test spans both the Java application server and the database server. The emulator and the benchmark driver must be deployed outside of the system under test so that they do not affect the benchmark results.

The benchmark driver executes five benchmark operations (cf. Figure 8.1). A dealer may `Browse` through the catalog of cars, `Purchase` cars, or `Manage` his dealership inventory, i.e., sell cars, or cancel orders. A manufacturer may place work orders for producing vehicles, which are triggered either over a web service interface (`CreateVehicleWS`) or via an RMI call (`CreateVehicleEJB`).

8.2.1.2. Hardware Setup

The benchmark application is deployed in the hardware environment depicted in Figure 8.2. We use seven blade servers from a local cluster environment. Each blade server is equipped with two Intel 4-core CPUs and 32 GB of main memory. The machines are connected by a 1 GBit Ethernet LAN.

On top of each machine, we run Citrix XenServer 5.5 as a virtualization layer. Inside the XenServer VMs, we run the benchmark components (application servers, driver agents, emulator, and load balancer). Each component runs in its own VM, initially equipped with two dedicated vCPUs. As operating system, these VMs execute CentOS 5.3. As Java EE application server, we use the Oracle WebLogic Server (WLS) 10.3.3. The load balancer is haproxy 1.4.8 using round-robin as load balancing strategy. The driver agents are provided by the Faban framework that is shipped with the benchmark. The database

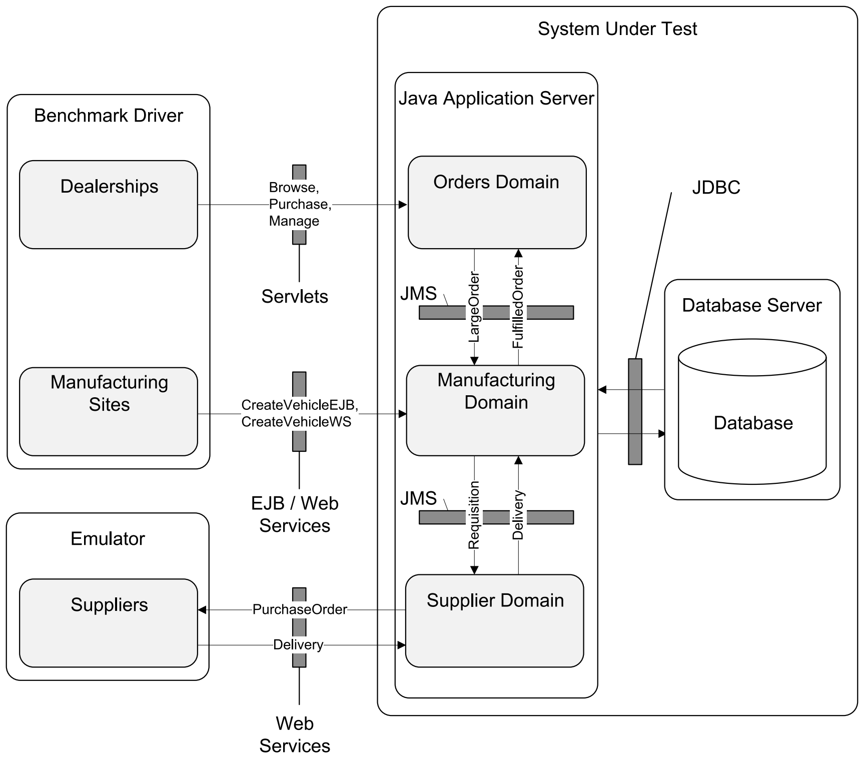


Figure 8.1.: SPECjEnterprise2010 benchmark architecture.

is an Oracle 11g database server instance deployed on a separate cluster node, hosting a dedicated VM with eight vCPUs running Windows Server 2008.

The SPECjEnterprise2010 benchmark application is deployed in this environment in an application server cluster of *WLS nodes*. Two types of dynamic changes are considered with respect to which the system is expected to adapt. First, the application usage profile changes periodically, i.e., the benchmark workload intensity and/or operation mix. Second, we have modified the benchmark such that it is also possible to activate or deactivate a service to emulate adding/removing services.

8.2.1.3. Degrees of Freedom

The system has the following degrees of freedom that we use to adapt the system to changes in the usage profile. First, we can add/remove WLS nodes to/from the WLS cluster. The load-balancer distributes the user requests equally among all WLS nodes that belong to the cluster. The minimum amount of cluster nodes is one, the maximum is restricted by the available hardware. The second and more fine-granular possibility to adapt the system is to add or remove vCPUs to/from a VM. Here, the minimum is two vCPUs, the maximum is four, as we require that all allocated resources are not shared among different VMs. Third, we can migrate VMs between the physical hosts. These adaptations are applicable at run-time, i.e., they can be applied while the benchmark application is running.

8.2.1.4. Resource Allocation Algorithm

To adapt the described system to changes in its environment, we need an adaptation process that performs adaptation actions to maintain performance requirements and resource efficiency. We now present a generic resource allocation algorithm that consists of two phases. 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. In the following, we give a formal description of this algorithm.

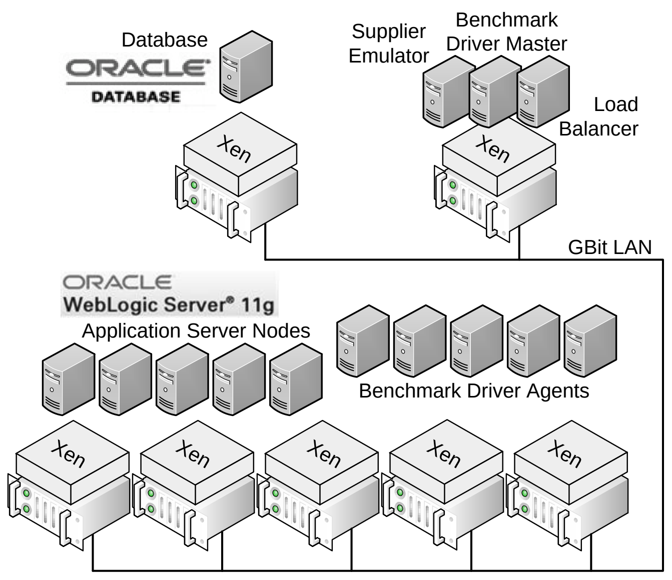


Figure 8.2.: Experiment setup.

Formally, we define the system as a 3-tuple $M = (T, S, C)$ where:

$T = \{t_1, t_2, ..., t_m\}$ is the set of resource types,
$S = \{s_1, s_2, ..., s_n\}$ is the set of services offered by the system,
$C = \{c_1, c_2, ..., c_l\}$ is the set of client workloads and respective SLAs. Each $c_i \in C$ is a triple (s, λ, ρ) where $s \in S$ is the service used, λ is the workload intensity (expected request arrival rate), and ρ is the client requested average response time (SLA).

Furthermore, we define the following functions:

$V \in [S \to 2^T]$ specifies which resource types are required by service $s \in S$,
$F \in [S \times T \to I^{s,t}]$, referred to as *resource allocation function*, assigns to each service $s \in S$ a set of instances $I^{s,t}$ of resource type $t \in T$ (e.g., a `Container` or `ContainerTemplate`, cf. Section 5.1.1). Each resource type instance is assumed to be allocated a given number of identical processing resources (cf. `ActiveResourceSpecifications` in Section 5.1.3). Formally, the resource type instance $i \in I^{s,t}$ is represented as a triple $(\pi, \kappa, \overline{\kappa})$, where π is the processing rate of its processing resources, κ is the number of processing resources currently allocated (e.g., allocated vCPUs), and $\overline{\kappa}$ is the maximum number of processing resources that can be allocated (e.g., number of CPUs on a physical machine),
$D \in [S \to \mathbb{R}^+]$ specifies the resource demand of service $s \in S$. If a service does not require a resource, the demand $D(s)$ is set to zero.

We define the following performance metrics:

$X(c)$ is the total number of requests of client workload $c \in C$ completed per unit of time (request throughput),
$R(c)$ is the average response time of a service request in client workload $c \in C$,
$U(t)$ is the average utilization of resource type $t \in T$ over all instances of the resource,
$\overline{U}(t)$ is the maximum allowed average utilization for resource type $t \in T$.

Finally, we define the following predicates:

$P_X(c)$ for $c \in C$ is defined as $(X(c) = c[\lambda])$,
$P_R(c)$ for $c \in C$ is defined as $(R(c) \leq c[\rho])$,
$P_U(t)$ for $t \in T$ is defined as $(U(t) \leq \overline{U}(t))$.

For a configuration represented by a resource allocation function F to be acceptable the following condition must hold $(\forall c \in C : P_X(c) \wedge P_R(c)) \wedge (\forall t \in T : P_U(t))$. For example, this condition can be checked in the ANALYZE phase to detect performance problems, or we can use it during the PLAN phase to evaluate if an adaptation action violates the performance requirements.

From a high-level perspective, the resource allocation algorithm is implemented as follows. Each time there is a change in the set of client workloads $C \longrightarrow \widetilde{C}$ (e.g., a new client workload $\widetilde{c} = (s, \lambda, \rho)$ is scheduled for execution or a change in the workload intensity λ of an existing workload is forecast), we use our online performance prediction mechanism to predict the effect of this change on the performance requirements expressed by the previously defined predicates. If an SLA violation is detected, the PUSH phase of our algorithm is executed which allocates additional resources until all client SLAs are satisfied. After the PUSH phase finishes, the PULL phase is executed to optimize the resource efficiency. If no SLAs are violated, the PULL phase starts directly. In the following, we describe the PUSH and PULL phases in more detail.

PUSH Phase

The following algorithm written in mathematical style pseudo code presents our basic heuristic for allocating resources to services such that client SLAs are satisfied.

Algorithm 1: PUSH

```
while $\exists c \in \widetilde{C} : \neg P_R(c)$ do
    forall the $t \in V(c[s]) : \neg P_U(t)$ do
        while $cap(c,t) \leq \overline{cap}(c,t)$ do
            if $\exists i \in F(c[s],t) : i[\kappa] < i[\overline{\kappa}]$ then
                $i[\kappa] \leftarrow i[\kappa] + 1$
            else
                $F(c[s],t) \leftarrow F(c[s],t) \cup \{\widehat{i}\}$
            end
        end
    end
end
```

Basically, while there exists a client response time SLA that is violated, the algorithm increases the amount of allocated resources for all resource types used by the service that currently exceed their maximum allowed utilization $\overline{U}(t)$. This is based on the assumption that violations are caused by at least one resource type used by the respective service has become a bottleneck. Increasing the number of allocated resources works as follows: If there is an instance of the overutilized resource type t (e.g., a VM) that has some unallocated processing resources available (e.g., virtual CPUs), additional resources are allocated. Otherwise, a new instance of the resource type $\widehat{i}$ is added (e.g., a new VM is started). In our algorithm, an additional resource instance increases the total capacity by one. Our algorithm assumes that there is an infinite amount of resources available and hence, this capacity increase is repeated until the amount of allocated resources reaches $\overline{cap}(\hat{c}, t)$, defined as

$$\overline{cap}(\hat{c}, t) = \left\lceil \frac{\sum_{c \in \widetilde{C}} c[\lambda] \cdot D(c[s])}{\sum_{c \in C} c[\lambda] \cdot D(c[s])} \right\rceil \cdot cap(\hat{c}, t), \tag{8.1}$$

where $cap(\hat{c}, t)$ is defined as the capacity assigned in the previous step. The above is an estimated upper bound on the capacity of resource type t required to handle client workload $\hat{c}$, based on a factor calculated by the changes in required resources (arrival rate times resource demand) and $cap(\hat{c}, t)$. It is calculated as the ratio of the newly specified arrival rates and the original arrival rates both multiplied with their respective resource demands. This estimation of the upper bound is intended to reduce the number of scenarios for which online performance prediction that have to be performed when searching for an acceptable configuration. Note that this is just one possibility to estimate an upper bound and other, more fine-grained estimates could be applied here as well.

PULL Phase

The PULL phase aims to optimize the resource efficiency by trying to release resources that are not utilized much by the current client workloads.

Algorithm 2: PULL

```
forall the $c \in C$ do
    $I \leftarrow \emptyset$
    while $\exists t \in V(c[s]) : \overline{U}(t) - U(t) \geq \epsilon \wedge I \neq F(c[s], t)$ do
        if $\exists i \in F(c[s], t) \backslash I : i[\kappa] > 0$ then
            $i[\kappa] \leftarrow i[\kappa] - 1$
            if $\exists c \in C : \neg P_R(c)$ then
                $i[\kappa] \leftarrow i[\kappa] + 1$
                $I \leftarrow I \cup i$
            end
            if $i[\kappa] = 0$ then
                $F(c[s], t) \leftarrow F(c[s], t) \backslash \{i\}$
            end
        end
    end
end
```

The algorithm is applied to all client workloads $c \in C$. While there is an instance of a resource type t assigned to service s of the currently considered workload c whose delta between the maximum utilization $\overline{U(t)}$ and current utilization $U(t)$ is greater than a predefined constant ϵ, the amount of resources allocated to this service will be decreased, i.e., for a resource type instance i of t which currently has some resources allocated (e.g., virtual CPUs), the amount of allocated resources is decreased. If the client SLAs are predicted to be violated after this change, the change is reversed. In case after the change, the instance has no remaining allocated resources, the instance i can be removed from the set of resource type instances (e.g., VM can be shut down). Note that the set of a resource type instances can also become empty, e.g., if there is no service left using the respective resource type t.

The resource allocation algorithm presented in this section is designed to find a system configuration that fulfills specified performance requirements. However, we do not claim that the resulting system configuration is optimal with respect to resource efficiency as the development of efficient optimization algorithms is not in the focus of this thesis.

8.2.2. Applied DML Instance

In this section, we introduce the different models that are part of DML instance we use to adapt the SPECjEnterprise2010 system to changing customer workloads. The adaptation process we model is the previously defined resource allocation algorithm.

8.2.2.1. Architecture-level Performance Model

DML-based architecture-level performance model that we use in this case study has been extracted automatically from a running benchmark deployment using the method presented by Brosig et al. (2011).

Briefly, the three main steps of the extraction process are: i) extraction of the application architecture, ii) extraction of the performance-relevant control flow, and iii) extraction of resource demands. The extraction uses monitoring data collected at system run-time with the Oracle WebLogic Diagnostics Framework (WLDF) running on the WLS instances. More details on the model extraction process can be found in Brosig et al. (2011).

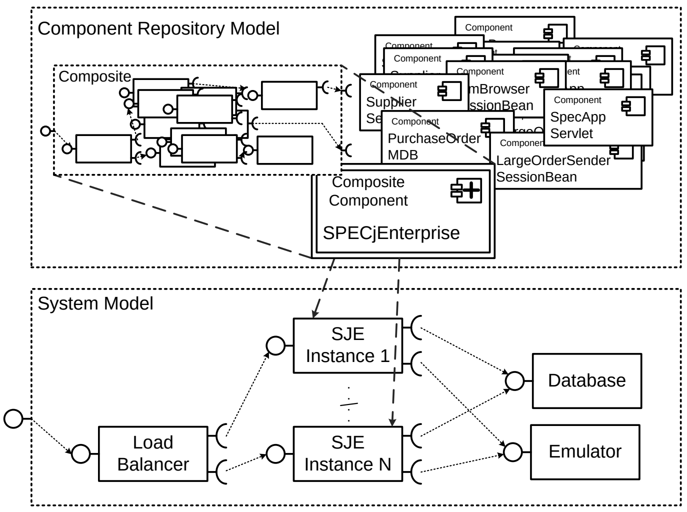


Figure 8.3.: Overview of the modeled SPECjEnterprise2010 benchmark components (cf. Brosig, 2014).

Figure 8.3 gives an overview of the resulting architecture reflected by the application architecture model of DML. The model shows a load balancer that distributes incoming requests to a cluster of of identical application servers each hosting an instance of the SPECjEnterprise2010 benchmark application, which themselves are connected to an emulator instance and a database instance. A benchmark application instance is a composite component which consists of several component instances, e.g., a `SpecAppServlet` component or a `PurchaseOrderMDB` component. These components reside in the repository as well. In total, the architecture-level performance model of the benchmark application consists of 28 components whose services are described by 63 fine-grained service behavior abstractions. In total, the model contains 51 internal actions, 41 branch actions, and four loop actions.

The usage model representing the benchmark workload was specified manually. Each of the five benchmark operations are modeled as individual usage scenarios. Figure 8.4 shows the usage scenario of benchmark operation *Manage* in a notation similar to UML activity diagrams. It consists of several system calls, two branches with corresponding transition probabilities, and a loop action. The loop iteration number is given as a probability mass function. For instance, in the depicted example, with a probability of 55% the loop body is executed only once, with a probability of 11% the loop iterates two times. The probabilities of the loop iteration numbers are derived from monitoring data. The remaining four benchmark operations are of similar complexity.

The resource landscape model describing the hardware environment in which we execute the SPECjEnterprise2010 benchmark was also specified manually. A model instance of the cluster nodes hosting one VM is depicted in Figure 8.6. The individual Containers are

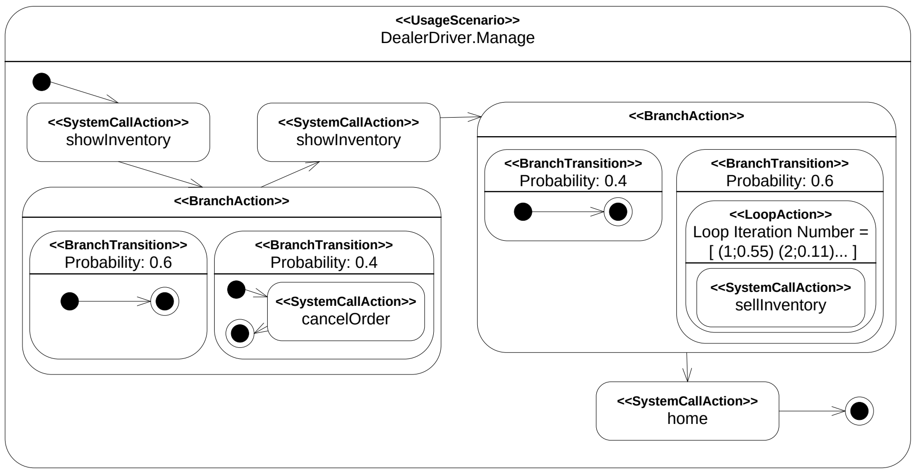


Figure 8.4.: Usage scenario for the *Manage* benchmark operation (cf. Brosig, 2014).

annotated with their respective ActiveResourceSpecifications. For example, the *DatabaseServer* is annotated with four ActiveResourceSpecifications representing four CPUs with four cores each.

Overall, the extracted model predicts the performance behavior very well, as evaluated for various scenarios presented in (Brosig et al., 2011). In summary, Figure 8.5 shows that resource utilization is predicted with a relative error of mostly 5% in a scenario with four application server nodes. Response times, which are typically much harder to predict accurately, are predicted with a relative error of about 10 to 20%. In this case study, we use the extracted architecture-level performance models to predict the impact of possible adaptations at run-time. A comprehensive and extensive evaluation of the accuracy of the employed performance models can be found in Brosig et al. (2011).

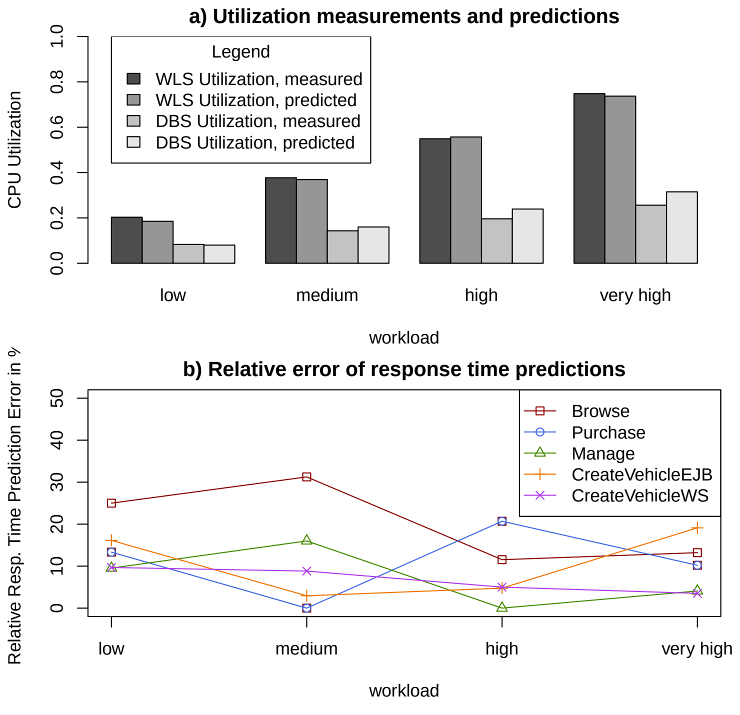


Figure 8.5.: Evaluation of the accuracy of the employed performance models (cf. Brosig, 2014).

8.2.2.2. Adaptation Points Model and Adaptation Process Model

As explained in Section 8.2.1.3, we have three degrees of freedom that we can use to adapt the SPECjEnterprise2010 system: the number of VM instances (WLS nodes), the number of vCPUs of a VM, and the host of a VM. These adaptation points and their respective boundaries are specified with the adaptation points meta-model. They are depicted together with the resource landscape model in Figure 8.6.

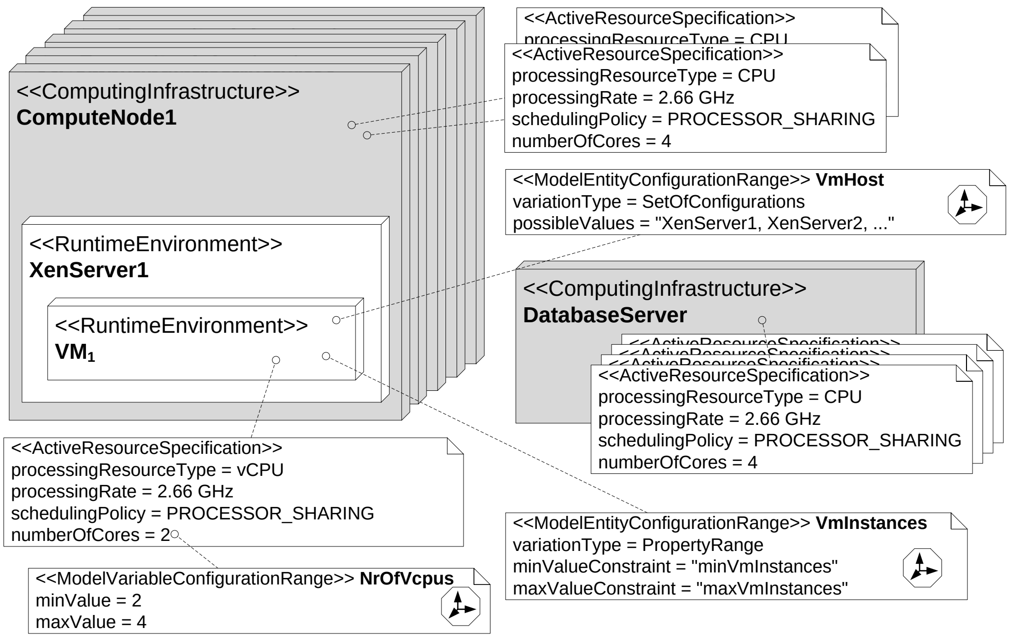

<<ComputingInfrastructure>>
ComputeNode1
<<RuntimeEnvironment>>
XenServer1
<<RuntimeEnvironment>>
VM1
<<ActiveResourceSpecification>>
processingResourceType = CPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 4
<<ModelEntityConfigurationRange>> VmHost
variationType = SetOfConfigurations
possibleValues = "XenServer1, XenServer2, ..."
<<ComputingInfrastructure>>
DatabaseServer
<<ActiveResourceSpecification>>
processingResourceType = CPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 4
<<ActiveResourceSpecification>>
processingResourceType = vCPU
processingRate = 2.66 GHz
schedulingPolicy = PROCESSOR_SHARING
numberOfCores = 2
<<ModelEntityConfigurationRange>> VmInstances
variationType = PropertyRange
minValueConstraint = "minVmInstances"
maxValueConstraint = "maxVmInstances"
<<ModelVariableConfigurationRange>> NrOfVcpus
minValue = 2
maxValue = 4

Figure 8.6.: Resource landscape and adaptation points of the SPECjEnterprise2010 benchmark deployment.

Based on the adaptation points, we have modeled an adaptation process using the adaptation process modeling language presented in Chapter 6. The modeled process is an instantiation of the resource allocation algorithm introduced in Section 8.2.1.4. More specifically, VMs correspond to a resource type $t \in T$ of the algorithm, and the number of vCPUs of a VM correspond to the capacity κ of a resource type instance.

Figure 8.7 shows a schematic representation of the adaptation process model instance and its strategies, tactics, and actions. The PUSH strategy is triggered if an SLA violation is detected, i.e., the predicate $P_R(c)$ evaluates to false. The strategy uses the IncreaseResources tactic to add resources until the problem is resolved, i.e., $P_R(c)$ is true. The way the tactic increases the amount of resources allocated to the application server depends on the capacity of the VM. If the number of assigned vCPUs is below the maximum ($cap(VM) < maxCap(VM)$), the control flow of the IncreaseResources tactic uses the addVCPU action to assign an additional virtual core to the VM. Otherwise, the tactic adds another VM to the application server cluster (addVM). The PULL strategy's purpose is to maintain resource efficiency. It is triggered after the PUSH strategy or on a scheduled basis, and removes resources until the SLAs start being violated again. Here, we can leverage the architecture-level performance model to predict if the application of a tactic would lead to an SLA violation. The PUSH strategy has three tactics to increase resource efficiency: RemoveVM, MigrateVM, or ReleaseVCPU. The different weights express which tactic to execute next. The weights of the tactics change during the adaptation of the system, depending on their performance and resource efficiency impact (cf. Section 6.2.5).

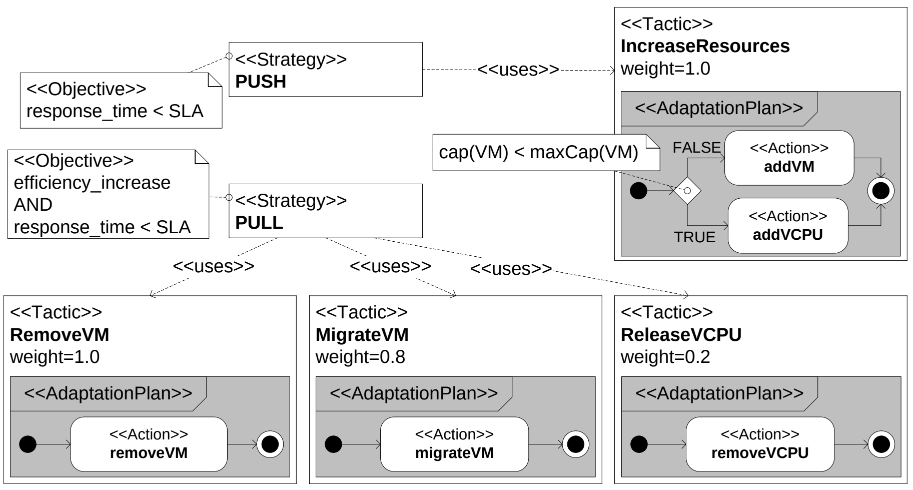


Figure 8.7.: A schematic representation of the adaptation process' strategies, tactics, and actions.

8.2.3. Evaluation

As part of the evaluation scenarios presented in the following, we specifically address the evaluation questions Q1 and Q3. We demonstrate that our approach is able to automatically adapt to the different changes in the system environment, such as adding new services or varying service workloads (Q1), and we evaluate the effectiveness of our approach compared to static resource provisioning (Q3).

8.2.3.1. Scenario 1: Adding a New Service

In this scenario, we evaluate if our approach is able to adapt the system in situations where new services are deployed in the environment on-the-fly such that performance requirements are maintained. Assume that there are four services executed in our environment and one application server node with two vCPUs is sufficient to handle the workload (default configuration c_0 in Figure 8.8). The SLAs for the different services correspond to the average response times we measured for the default arrival rates settings of the benchmark driver. Using the tuple (*service_name*, *arrival_rate*, *response_time*) as notation for SLA specification, the SLAs for the four currently running services are: (`CreateVehicleEJB`, 15, 54ms), (`Purchase`, 12.5, 80ms), (`Manage`, 12.5, 80ms), and (`Browse`, 25, 80ms).

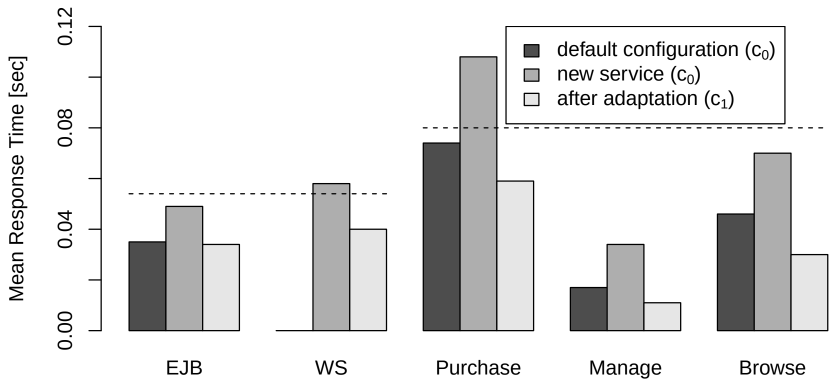


Figure 8.8.: The response times of the five services and their respective SLAs (denoted by the dashed lines) before and after adapting the system.

Now a new service with the SLA (`CreateVehicleWS`, 15, 54ms) is added. After adding the new service to the model, the ANALYZE phase detects SLA violations for the services `CreateVehicleWS` and `Purchase` (cf. Figure 8.8). To resolve this problem, the modeled adaptation process is triggered to find a new system configuration that can maintain the SLAs. The adaptation process starts with the PUSH strategy, trying to increase the resources in the WLS cluster. As the currently active VM has some capacity left (it uses only two vCPUs), the IncreaseResources adds an additional vCPU to the existing VM (configuration c_1). After adapting the model, the analysis of the model indicates satisfied SLAs, i.e., a resolution of the problem that triggered the adaptation. To maintain resource efficiency, an additional adaptation strategy is now triggered that checks if the system can be further reconfigured to increase resource efficiency while maintaining SLAs. In this case, nothing can be done and therefore, the resulting configuration recommended by the modeled adaptation process consists of one node with three vCPUs. To confirm these results, we have conducted experiments measuring the service response times in the different configurations. The results are depicted in Figure 8.8 and show that with the default resource allocation, the SLAs of `CreateVehicleWS` and `Purchase` cannot be maintained. However, after applying our modeled adaptation process, all SLAs are satisfied. This experiment also shows that for fine-granular changes such as adding a single service, we need fine-granular models and analysis techniques to find the proper system configuration.

8.2.3.2. Scenario 2: Increasing Workload

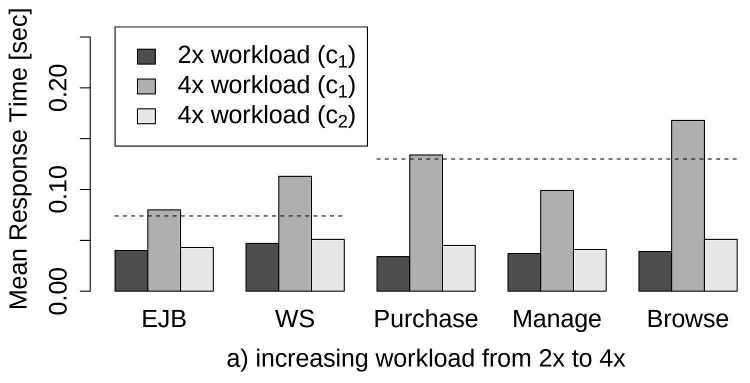
In this scenario, we evaluate our approach when the workload of all services deployed in our environment increases. We increase the workload in two steps, from 2x to 4x, and from 4x to 6x (see Figure 8.9). The standard workload (1x) is the workload as defined in the previous scenario for all five services.

Our starting point is that five services are running on one node (c_1) with twice the standard workload and the following SLAs (`CreateVehicleEJB`, 30, 74ms), (`CreateVehicleWS`, 30, 74ms), (`Purchase`, 25, 130ms), (`Manage`, 25, 130ms), and (`Browse`, 50, 130ms), which are all initially satisfied. Now, we increase the workload to 4x the standard load. For this new workload, we can detect a violation of the SLAs. After applying the adaptation process leads to a system configuration with two VMs, one with four vCPUs and one with three vCPUs (c_2). Applying this reconfiguration to our experiment environment, the measurement results depicted in Figure 8.9 a) confirm that all SLAs are maintained, i.e., the adaptation was successful.

In the second step, we increase the workload to 6x the standard load, not changing the SLAs. Again, this leads to a violation of the SLAs. Now, the adaptation process recommends to add a further VM, i.e., we now use two VMs with four vCPUs and one VM with three vCPUs (c_3). However, our measurements depicted in Figure 8.9 b) show that after reallocation the SLAs of the `Browse` and the `Purchase` services are still violated. A detailed analysis of the SPECjEnterprise2010 component response times revealed that the reason is the database, which was not powerful enough to handle the new workload. Hence, we have to scale the database to investigate if the recommended configuration is valid. This is explained in the following section.

8.2.3.3. Scenario 3: Scaling the Database

To scale the database, we moved the DBMS to a more powerful machine with four 6-core AMD CPUs and 128 GB main memory and connected the machine to our cluster environment with four 1 Gbit Ethernet links. Furthermore, the DBMS was deployed on a VM but on a native Windows 2008 Server. With this setup, the DBMS was able to handle the increased load and we can confirm that configuration c_3, that was recommended by



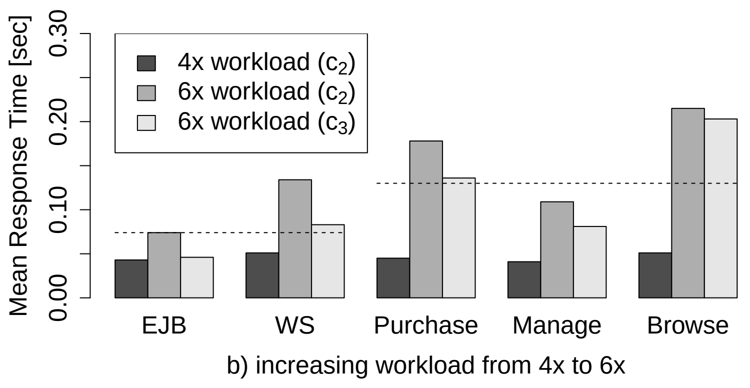


Figure 8.9.: The response times when increasing the workload intensity from 2x to 4x and 4x to 6x, respectively (SLAs denoted by the dashed lines).

our adaptation process, is sufficient to maintain the SLAs if we increase the workload to 6x the standard workload (cf. Figure 8.10).

Next, we tried to further scale the workload from 6x to 8x. With this configuration, we again detect SLA violations and the adaptation process suggested to add one further node (c_4). However, in the real system we measured only an insignificant reduction of the response times with this new configuration (cf. Figure 8.10). This time the problem was the load balancer which was fully utilized, busy with handling over 5500 sessions per second. Our approach was not able to detect this problem since we have not considered the load balancer in our application-level performance model. However, even though the load balancer can be easily integrated into our model, this would not help to further scale the experiment environment because the model's analysis results would then also limit the size of the application server cluster to four nodes. To further increase the size of the application server cluster, one would have to reconfigure or scale the load balancer performance to to increase its throughput.

8.2.3.4. Scenario 4: Decreasing Workload

This scenario evaluates how our approach adapts resource allocations in situations when the workload decreases. In such scenarios, the approach should release resources that are not needed while ensuring that SLAs are not violated. This improves system efficiency because the released resources can then be assigned to other VMs (in case of released vCPUs) or one can switch a whole physical machine into standby mode in case all its VMs are released or migrated.

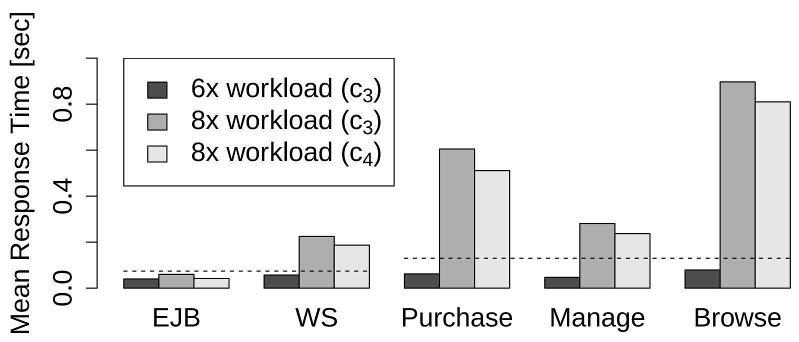


Figure 8.10.: The response times when increasing the workload intensity from 6x to 8x with the DBMS deployed on a more powerful machine. SLAs are denoted by the dashed lines.

Assume the situation that all services are executed with 6x the standard workload on three nodes and all SLAs are satisfied (c_3). Now, we decrease the workload to 5x the standard load. In the Analyze phase, we can detect the workload decrease and trigger the Plan phase to execute an adaptation. In this scenario, the adaptation process recommends a configuration c_2 in which two nodes are sufficient to handle the decreased workload. The measurement results are depicted in Figure 8.11, demonstrating that the recommended system configuration is correct, as configuration c_2 requires less resources but still maintains the SLAs.

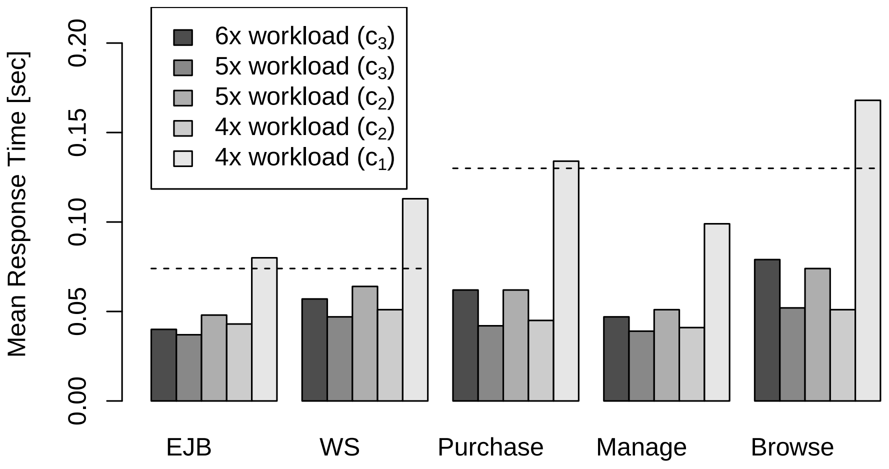


Figure 8.11.: The response times when decreasing the workload intensity from 6x to 5x to 4x before and after adaptation and with further decreased resources (SLAs denoted by the dashed lines).

Next, we decreased the workload to 4x the standard load. Again, our approach predicts that two nodes (c_2) are sufficient to handle the decreased workload. To confirm that resource allocations cannot be further reduced while satisfying the SLAs, we conducted an experiment where we further reduced the allocated resources manually to one node (c_1). The results for this configuration show that it would violate the SLAs, hence the previously found configuration is valid.

8.2.3.5. Scenario 5: Migration of Virtual Machines

In the previous scenario, we have shown that the system efficiency can be improved by adjusting the number of allocated resources to match the current workload intensity. However, when reducing resources by removing application server nodes (stopping VMs), it is possible that some of the physical machines are not utilized efficiently. For example, one configuration might consist of two 8-core machines, each running one VM with 4 cores, i.e., each machine is not fully utilized. Therefore, our adaptation process considers migrating VMs to consolidate them on fewer physical machines in order to further increase the resource efficiency. For this scenario, assume that we have just experienced a workload decrease from 6x standard workload to 4x and the system now runs with two application server nodes, each on a separate physical machine. Now, through VM live migration, our approach can consolidate these two VMs on a single machine.

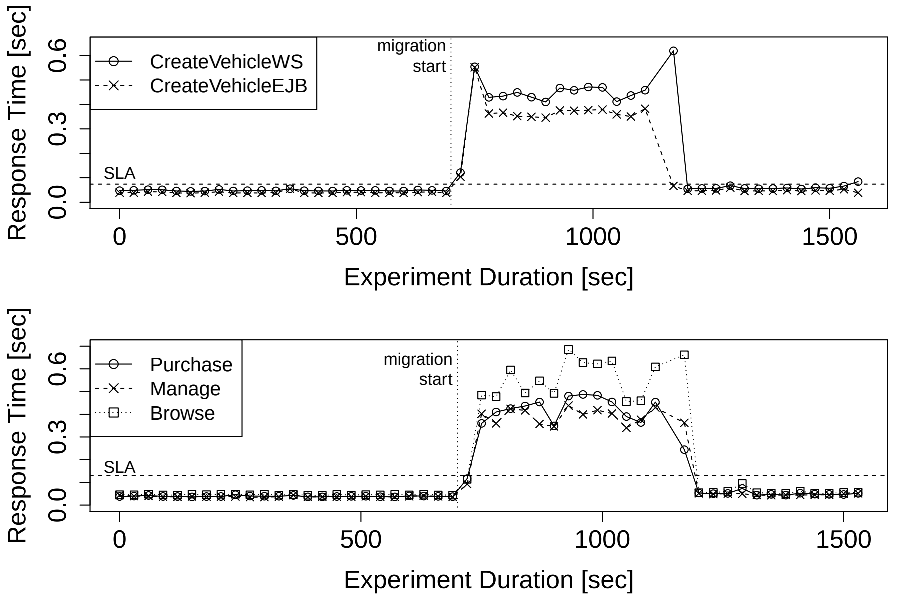


Figure 8.12.: Response time measurements during migration of an application server node.

Figure 8.12 shows the effect of the migration on the services' response times. We can observe a significant increase of the response time for all services directly after starting the migration which was triggered 700 seconds after the begin of the experiment. The response times remain increased as long as the VM is being migrated (approx. 500 seconds). In a similar experiment under light load conditions, the migration takes only approximately two minutes. Therefore, the time taken to migrate the VM depends on the load of the migrated VM. Hence, when migrating a VM, our approach must trade-off penalties due to violated SLAs with the benefits of increased efficiency, an aspect that we evaluate in the next subsection.

8.2.3.6. Scenario 6: Resource Usage and Efficiency

To illustrate the potential efficiency gains achieved by our approach, we use real-life workload data of a mainframe provided by our industrial partner IBM (Herbst et al., 2013a). The data, shown in Figure 8.13, reports the number of started transactions during one week from Monday to Sunday in 15 minute time frames (in total 575 frames). We assume that the maximum of this workload corresponds to our maximum of 8x the standard workload.

With our approach, we can dynamically adapt the amount of allocated resources to the changing workload. The bottom curve of Figure 8.13 depicts the development of the

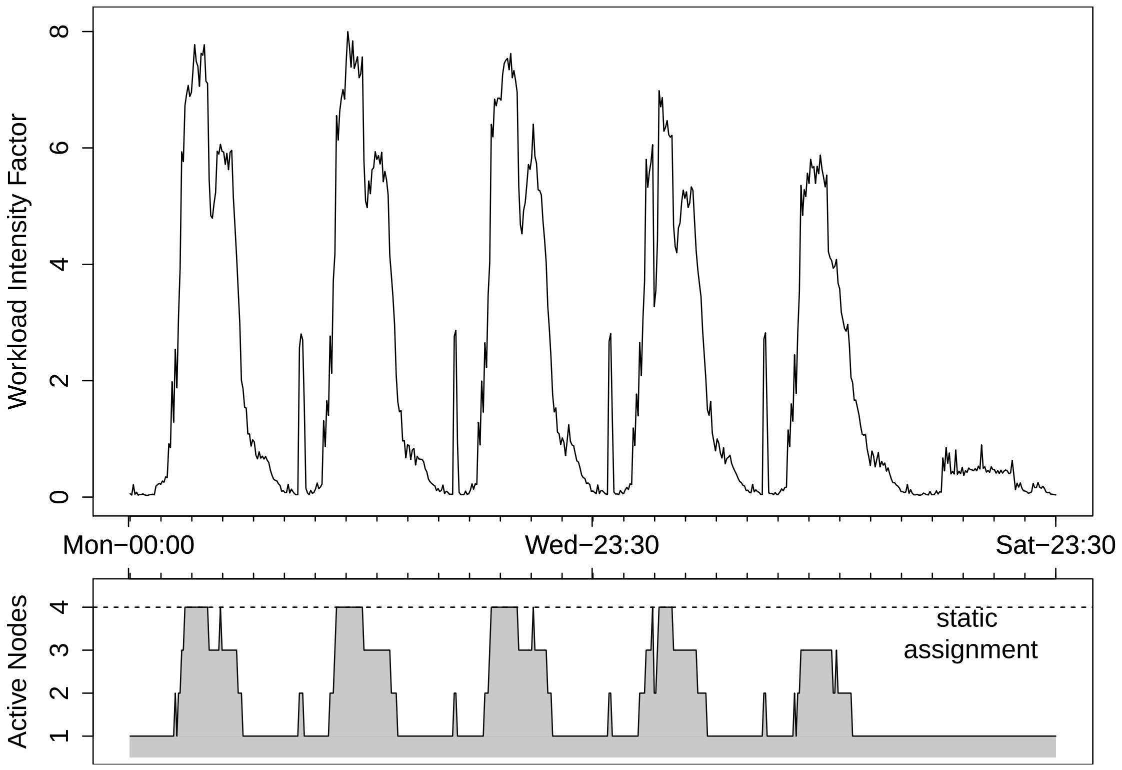


Figure 8.13.: Active nodes for a workload distribution over six days.

allocated resources if we apply the presented adaptation process to assign and release resources. In a static scenario, one would assign four dedicated nodes to guarantee SLAs in peak load situations, depicted by the dashed line in the bottom of Figure 8.13. Specifically, one would use four nodes for all 575 time frames, i.e., 2300 active nodes in total. With a dynamic resource allocation approach, only $\sum_{i=1}^{575} active\_nodes(i) = 1002$ active nodes, i.e., 43.57% of the resources of the static assignment are needed. Thereby approximately 56% of the resources available can be released. These resources can be either used for other customers or applications, or the application can be consolidated to free physical machines and switch them to standby mode. Given that the monitoring time frames are 15 minutes and the migration of a VM takes only around seven minutes, we can achieve improved efficiency for approximately eight minutes.

In this case study, we used our approach to adequately react on changes in the system environment to adapt the system accordingly. However, the approach presented in this thesis is also capable of proactive system adaptation. Thus, to evaluate the proactiveness of our approach, the following case study demonstrates how we can use workload forecasting techniques to anticipate performance problems and adapt the system before such problems actually occur.

8.3. Proactive Model-Based System Adaptation

In Chapter 7, we presented an approach for self-adaptive Workload Classification and Forecasting (WCF) at run-time. WCF uses well-established time series analysis techniques to identify the characteristics of workloads such that we are able to automatically select suitable forecasting methods to estimate future workload intensities. Due to the large spectrum of integrated forecasting methods, WCF offers a high degree of flexibility to influence the forecast accuracy by adapting the selected forecasting methods to the given workload intensity behavior (WIB). The experiment results in Section 7.4 have shown that WCF is able to select appropriate forecasting methods, thereby improving the overall forecast accuracy significantly. Nevertheless, the question remains if WCF is capable of forecasting the workload behavior with sufficient accuracy such that the forecasts are useful for proactive performance and resource management at run-time.

In this section, we present two scenarios to demonstrate how WCF can be used to reduce SLA violations and for proactive resource provisioning at run-time (Validation Question Q2). First, we will apply WCF to a typical web server workload and evaluate the quality of the forecasts to avoid SLA violations. The adaptation behavior of the approach presented in the previous case study was only reactive. Thus, the second scenario evaluates the applicability of WCF for proactive resource provisioning at run-time by applying it to the transactional workload of the SPECjEnterprise2010 case study presented in Section 8.2. Simultaneously, we evaluate the effectiveness of our approach compared to static and reactive adaptation approaches (Validation Question Q3).

8.3.1. Scenario 1: Proactively Reducing SLA Violations with WCF

For the following experiment, we assume that there is a system consisting of three web servers responsible for handling a variable amount of page requests. We assume that the system is linearly scalable from one to three server instances, i.e., the average resource demand per request is constant for all three server instances. Furthermore, we assume that the considered system implements a trigger-based resource provisioning approach that reacts on observed SLA violations. If the arrival rate, i.e., the number of page requests per hour grows above a certain threshold, the SLA of the average response time is assumed to be violated and the system adds an additional server. Similarly, if the arrival rate decreases below a certain threshold, we assume that the resource efficiency is violated and the system releases a running server. In the following, we use the term SLA violations also for resource efficiency violations.

In this scenario, we apply our WCF approach to forecast the workload, i.e., the page requests from the web servers. The goal of this scenario is to demonstrate that we can use our WCF approach to forecast the arrival rates of the system. With these forecasts, we can then proactively add or release server instances to avoid SLA violations. As a representative workload, we use the number of page requests of Wikipedia Germany obtained from the Wikimedia Project (2011). We assume that WCF has no historical knowledge about the Workload Intensity Behavior (WIB). To achieve accurate forecast results, WCF has to identify the WIB characteristics and select applicable forecasting methods from the four overhead groups presented in Section 7.2.2. A summary of the WCF configuration for this scenario is given in Table 8.1.

In Figure 8.14, the WCF forecast values are plotted together with the corresponding confidence intervals and the observed workload. On the x-axis, we show the forecasting method that has been chosen by WCF in the respective period to forecast the next h time series points. The two dotted lines represent the given thresholds that define when a server instance is added or released. To generate some variation in the amount and occurrence of SLA violations, the upper threshold is placed such that it is not reached in every daily seasonal period, e.g., not on the weekends. If we look at the quality of the forecasts to anticipate SLA violations, i.e., an intersect of the forecast time series with one of the thresholds, we can see that SLA violations cannot be reliably anticipated in the first three daily periods. The reason is the relatively little information about the WIB that is available to classify and forecast the workload. For the following daily periods, as the amount of available information grows, we can reliably anticipate SLA violations in the majority of cases. Only when the amplitude of the daily pattern changes, e.g., before and after weekends, the forecast mean values deliver false positives or do not anticipate the need for additional computing resources in time.

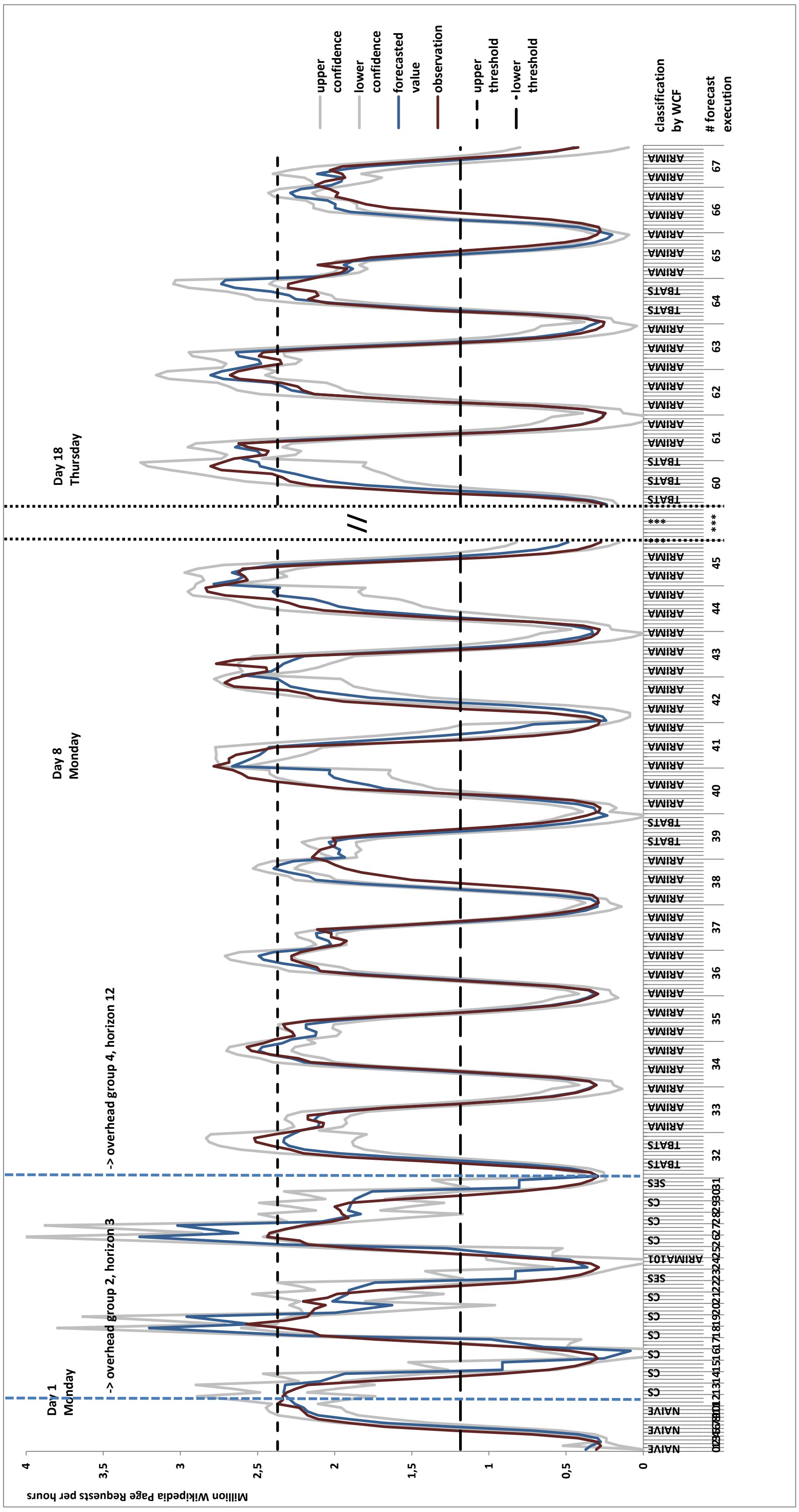


Figure 8.14.: Observed workload (Wikipedia page requests) and the forecasts of WCF for 21 days.

Table 8.1.: Settings of WCF.

Forecasting Method (overhead group)	WCF(1-4)
Input Data WIB trace	Wikipedia Germany, 3 weeks, 504 values in page requests per hour, frequency = 24, 21 periods (days)

Forecast Horizon (= Forecasting Frequency) (number of forecast points h)	h = 1 for 1^{st} half period
	h = 3 until 3^{rd} period is complete
	h = 12 from 4^{th} period

Confidence Level	80%

To evaluate the benefit of our WCF approach for proactive resource provisioning, we compare the amount of SLA violations between a reactive approach based on the monitored workload and a proactive approach using WCF. In the reactive case, adding and removing of server instances is triggered when the monitored workload reaches the specified threshold. Thus, every intersection of the observed workload time series with one of the threshold lines indicates an SLA violation. In total, we count 76 SLA violations listed in Table 8.2.

Table 8.2.: Summary of the SLA violations of the reactive and proactive approaches.

Approach	**Result**
Reactive	76 SLA violations
Proactive (using WCF)	42 SLA violations correctly anticipated plus 15 almost correct anticipations 6 cases of false positives 13 cases of false negatives (not detected)

To quantify SLA violations for the proactive approach using WCF, we first have to count the number of correct, incorrect and "almost correct" forecasts for all threshold intersections of the observed workload. With the term "almost correct" forecasts we refer to intersections with the thresholds where the accurate value was forecast slightly too late or too early compared to the configured forecast horizon (cf. Table 8.2). If the forecast value is correct, i.e., it corresponds to the observation, the threshold intersection has been predicted accurately. If the forecast value is incorrect, i.e., it indicates a threshold intersection although there is actually no intersection, this could lead to either a false negative or a false positive SLA violation. The false positives lead to predicted SLA violations when there are actually no violations, whereas the false negatives refer to situations where there has been an SLA violation that was not anticipated. Table 8.3 provides a detailed list of the SLA violations we counted for our forecasts. The term "reactive" in Table 8.3 indicates that the threshold intersection forecast was too late, i.e., an SLA violation occurs which must be handled reactively. As "proactive" we denote threshold intersections that are forecast correctly, i.e., we can proactively adapt the system to avoid SLA violations.

In summary, in the best case, i.e., if we count the almost correct forecasts as correct, 57 out of 76 (75%) of all SLA violations of the reactive approach can be avoided (cf. Table 8.2). In the worst case, we have to accept 6 false positives, 13 false negatives, and 15 incorrect anticipations, but we can still avoid 55% of the SLA violations of the reactive approach.

Table 8.3.: Detailed list of the SLA violations in the reactive and proactive approaches.

Day	Lower Threshold: 1 or 2 Server Instances			Upper Threshold: 2 or 3 Server Instances		
	Direction	**Action Type**	**Comment**	**Direction**	**Action Type**	**Comment**
1	up	reactive	not detected	n/a	none	
1	down	reactive	not detected	n/a	none	
2	up	reactive	not detected	up	proactive	short prov. time
2	down	reactive	not detected	down	reactive	not detected
3	up	reactive	not detected	up	proactive	short prov. time
3	down	reactive	not detected	down	reactive	not detected
4	up	proactive		up	reactive	not detected
4	down	proactive		down	reactive	not detected
5	up	proactive		up	proactive	
5	down	proactive		down	proactive	
6	up	proactive		up	proactive	false positive
6	down	proactive		down	proactive	false positive
7	up	proactive	slightly too early	n/a	none	
7	down	proactive	slightly too early	n/a	none	
8	up	proactive		up	reactive	not detected
8	down	proactive	slightly too late	down	proactive	
9	up	proactive	slightly too late	up	proactive	slightly too late
9	down	proactive		down	proactive	slightly too early
10	up	proactive		up	proactive	slightly too late
10	down	proactive		down	proactive	
11	up	proactive		up	proactive	
11	down	proactive		down	proactive	
12	up	proactive		up	proactive	
12	down	proactive		down	proactive	
13	up	proactive		up	proactive	false positive
13	down	proactive		down	proactive	false positive
14	up	proactive	slightly too early	n/a	none	
14	down	proactive	slightly too early	n/a	none	
15	up	proactive		up	reactive	not detected
15	down	proactive		down	proactive	
16	up	proactive		up	proactive	slightly too late
16	down	reactive	not detected	down	proactive	slightly too early
17	up	proactive		up	proactive	slightly too late
17	down	proactive		down	proactive	
18	up	proactive		up	proactive	
18	down	proactive		down	proactive	
19	up	proactive		up	proactive	
19	down	proactive		down	proactive	
20	up	proactive		up	proactive	false positive
20	down	proactive		down	proactive	false positive
21	up	proactive	slightly too early	n/a	none	
21	down	proactive		n/a	none	

8.3.2. Scenario 2: WCF for Proactive Resource Provisioning at Run-Time

In this scenario, we apply WCF to forecast the transactional workload of the SPECjEnterprise2010 case study presented in Section 8.2. The purpose is to demonstrate the potential efficiency gains achieved by of our proactive model-based system adaptation. Therefore, we compare the total amount of allocated resources and the number of SLA violations of our proactive approach with the static and reactive resource allocation approaches presented in Scenario 6 in Section 8.2.

To realize proactive system adaptation, we use the workload forecasts to adjust the usage profile model of SPECjEnterprise2010. Then, we use the online performance prediction techniques by Brosig (2014) to determine the impact of the forecast workload change on the service response times. If we anticipate an SLA violation, we start our system adaptation approach to allocate more resources such that the SLAs are maintained. Similarly, if we predict a workload decrease, we can leverage the architecture-level performance model to analyze if we can reduce resources without violating SLAs.

As a representative workload, we use the the same workload as in Section 8.2, a transactional workload provided by IBM from a real-life deployment of an IBM z10 mainframe server (Herbst et al., 2013a). As opposed to the previous scenario, we now assume that WCF has historical knowledge about the WIB which can be used to select appropriate forecasting methods. A summary of the WCF configuration for this scenario is given in Table 8.4.

Table 8.4.: Settings of WCF.

Forecasting Method (overhead group)	WCF(1-4)
Input Data WIB trace	CICS, Monday to Saturday, 576 values in transactions per 15 minutes, frequency = 96, 6 periods (days)

Forecast Horizon
(number of forecast points h)
	h = 1 for 1^{st} half period
	h = 2 until 6^{th} period is complete

The top of Figure 8.15 depicts the workload we used before in the SPECjEnterprise2010 case study (solid line) in comparison to the forecasts of WCF (dashed line). Recall that the considered workload is the number of started transactions during one week from Monday to Sunday in 15 minute time frames (in total 575 frames).

It shows that the forecast values fit the actual workload quite well except for the peaks appearing between the days. At the bottom of Figure 8.15, we plotted the number of allocated server nodes. The gray area delimited by the solid line depicts the course of the assigned nodes if we use the reactive approach as presented in the SPECjEnterprise2010 case study in Section 8.2. If we use the forecasts for proactive resource allocation based on our model-based adaptation approach, we observe a different development of allocated resources, depicted by the difference between the solid and the dashed lines.

To quantify the differences between the static, reactive and proactive resource allocation approaches, we compared the total amount of allocated resources as well as the number of resulting SLA and resource efficiency violations.

In the static resource allocation approach, we assigned four dedicated nodes to guarantee SLAs even in peak load situations, i.e., we experience zero SLA violations. However, in this case, four nodes are active for all 575 time frames, i.e., 2300 active nodes in total.

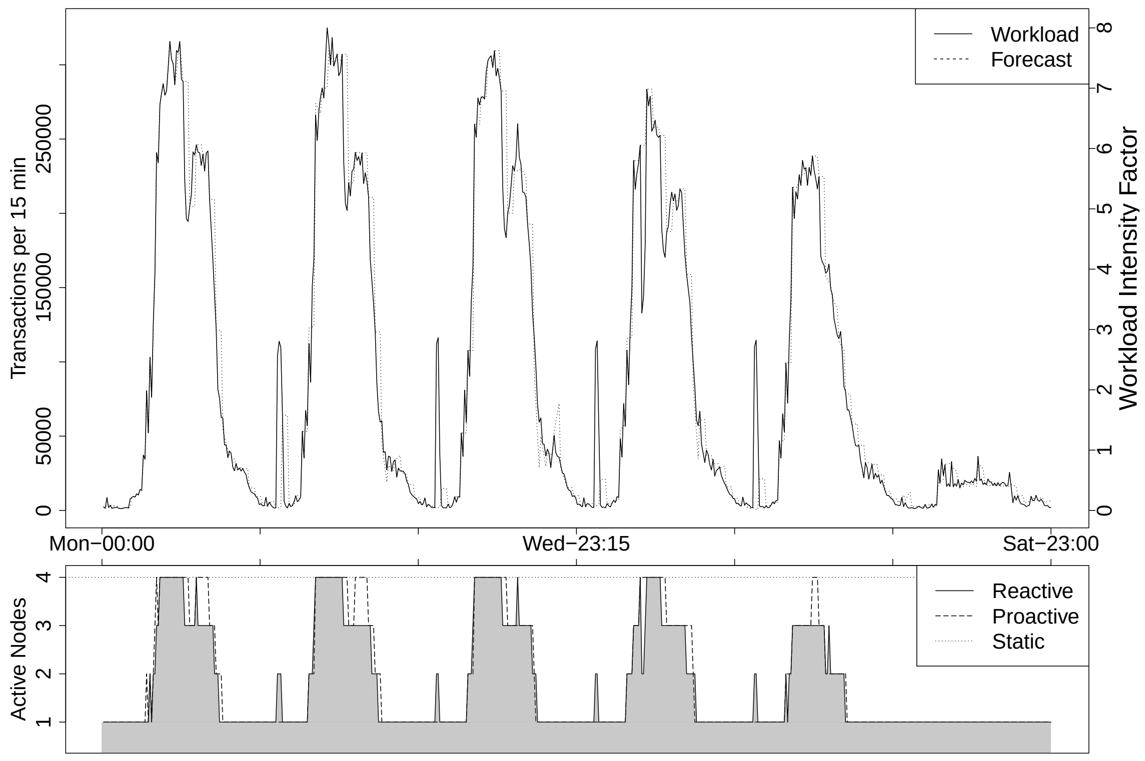


Figure 8.15.: Active nodes for a workload distribution over six days using a static, reactive, and proactive approach.

In the reactive case, we perform an adaptation action (adding or removing a node) if an SLA is violated or if a resource is not utilized efficiently (cf. Section 8.2). During the considered six days, the reactive approach allocated a total amount of 1002 active nodes and performed 109 adaptation actions (cf. Table 8.5). Out of these 109 adaptation actions, in 52 cases, nodes were added due to SLA violations, and in 57 cases, one or multiple nodes were removed to increase resource efficiency. Thus, with the reactive approach, only 44% of the resources of the static assignment are needed. However, this advantage comes at the cost of some SLA violations.

In the proactive case, we count a total resource allocation of 1040 servers and 43 SLA and resource efficiency violations. As a result of this experiment, we can say that proactive resource provisioning needed approximately 5% more resources than the reactive approach (1002 vs. 1040), but avoided approximately 60% of the SLA and resource efficiency violations of the reactive approach. Regarding the applicability of WCF, this experiment shows that in almost 20% of the cases, the forecast was too late, which resulted in an SLA violation. However, this could be further improved by choosing more suitable time intervals for forecasting and adaptation.

Table 8.5.: Summary of the SLA and resource efficiency violations as well as the amount of allocated resources for the static, reactive, and proactive approaches.

Approach	**SLA and efficiency violations**					**Allocated**
	Too late	**Too early**	**False pos.**	**False neg.**	**Total**	**Resources**
Static					0 (–)	2300 (100%)
Reactive					109 (100%)	1002 (43.57%)
Proactive	20	6	5	12	43 (39.45%)	1040 (45.22%)

As a result, we conclude that the reactive and proactive model-based system adaptation approaches save approximately the same amount of resources (cf. Table 8.5). However, the important difference is that the proactive approach, compared to the reactive approach, significantly reduces the amount of SLA violations.

8.3.3. Evaluation

Our previous experiments, presented in Section 7.4, showed that WCF can significantly improve the forecasting accuracy with acceptable additional computational overhead. The experiments presented in this section additionally demonstrated the applicability of the forecasts obtained with WCF for proactive resource provisioning at run-time. In the first scenario, we used WCF to forecast typical web server workloads, and analyzed if the forecasts can be used to predict SLA violations. In the second scenario, we applied WCF to the transactional workload of our SPECjEnterprise2010 case study presented in Section 8.2. We then used the workload forecast in our model-based approach to proactively adapt the system resources to changes in the workload. In summary, the results show that with WCF, we are able to reduce the number of SLA violations by approximately 55% to 75%. However, this comes at the cost of approximately 5% increased overall resource usage compared to a reactive, trigger-based approach. Thus, we conclude that WCF can support proactive model-based performance and resource management.

8.4. Model-Based System Adaptation in Heterogeneous Environments

In this case study, we apply our approach in a heterogeneous resource environment provided by our industrial partner Blue Yonder. Blue Yonder is a leading service provider in the field of predictive analytics and big data. The company offers enterprise software services that are based on forecasts of, e.g., enterprise sales, costs, or churn rates. Blue Yonder employs machine learning techniques to obtain accurate forecasts based on historical data provided by their customers. Usually, supervised machine learning can be applied, consisting of a training step that is used to infer a mathematical model from the available historical data. This model can then be used to calculate forecasts based on a given input data set. Training the model and calculating the forecasts requires a considerable amount of computational resources depending on the amount of customers, their input data, and their Service-Level Agreements (SLAs).

Currently, Blue Yonder uses dedicated resources for each customer to fulfill their respective SLAs. When acquiring new customer projects, Blue Yonder normally has to estimate how much resources are required to sustain the workloads of the new customers and ensure adequate performance. This estimation is based on the experience of Blue Yonder's employees and can range from few low-budget desktop machines to hundreds of cores on high-end servers, depending on the customer's amount of data to be analyzed and on the time available for the analysis. More importantly, this estimation is generally a worst-case estimation, i.e., the system capacity is dimensioned to support the peak workload intensity.

Given the increasing number of servers and respective operating costs, Blue Yonder is interested in increasing resource efficiency by sharing resources among different customers. As Blue Yonder has detailed information from their customers about when, how many, and which type of requests are expected to arrive (*request schedule*), a self-adaptive approach that assigns the required amount of resources to new customers and dynamically adapts the amount of resources according to the actual customer demand appears to be promising.

The major goal of this case study is to evaluate if our approach is applicable in Blue Yonder's scenario and if it is capable of increasing the system's resource efficiency (Validation

Question Q1). Additionally, it is crucial that the adaptation is capable of considering the different performance requirements of Blue Yonder's customers. A particular challenge of this case study is that our approach is faced with a heterogeneous hardware environment—low-cost desktop computers and high-end servers—and with different performance requirements of multiple concurrent customers. Thus, a further research question targeted here is whether our approach is applicable in an environment with heterogeneous resources and whether it can be effectively used to trade-off different performance requirements of multiple customers (Validation Question Q4).

In the following, we first explain the architecture of Blue Yonder's system and present the respective architecture-level performance model, modeled with the Descartes Modeling Language (DML). Next, we specify an adaptation process using DML's adaptation process model to adapt the system to changes in the environment considering the previous requirements. Finally, we present the results when applying our model-based self-adaptive performance and resource management approach in Blue Yonder's system.

8.4.1. Blue Yonder System Architecture

A typical Blue Yonder system consists of three main software component types: the Gateway Server (`GW`), the Prediction Server (`PS`), and a third party component, the database (`DB`). These software components can be distributed in a heterogeneous resource environment, e.g., as depicted in Figure 8.16. The `GW` is the communication endpoint to the Blue Yonder system. Users can invoke a set of different services via HTTP. In the considered sample project, the available services are `train`, `predict` and `results`. As their names suggest, the `train` service initiates the training step of the supervised learning algorithms. The `predict` service initiates the calculation of the forecasts using the trained prediction model, and finally the `results` service provides the results to the customer. To train the prediction model, the `train` service accepts historical data. The `GW` receives this data, parses it and generates a job, which is put into the `GW`'s queue and scheduled for processing. Then, an active `PS` takes the job from the queue, processes it (i.e., trains the prediction model) and stores the results in the database. After training, a user can invoke the `predict` service to calculate a forecast based on the trained prediction model. The user sends the data for which the forecast should be made to the `GW`. The `GW` reads the data and generates one or several jobs—depending on the size of the data—which are scheduled for processing. These jobs are again processed by one or several `PS` and the results are stored in the database for retrieval by the user (`results` service). Technically, `GW` and `PS` are independent operating system processes that can be started and stopped on any machine in the resource landscape. The database for our case study is a standard MySQL database. Each customer has its own `GW`, `PS`, and `DB` instances, which are deployed in Blue Yonder's resource landscape. The number of component instances and their distribution in the system environment is called *topology*.

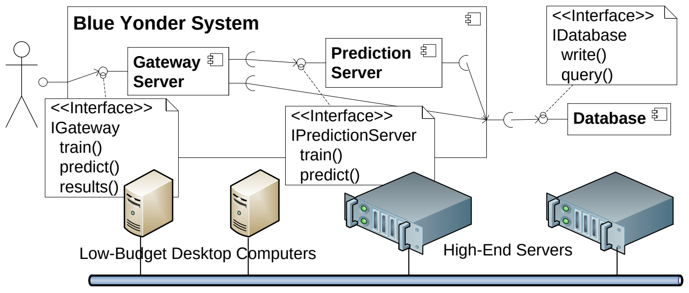

Blue Yonder System
Gateway Server
Prediction Server
<<Interface>> IGateway train() predict() results()
<<Interface>> IPredictionServer train() predict()
<<Interface>> IDatabase write() query()
Database
Low-Budget Desktop Computers
High-End Servers

Figure 8.16.: Topology of an example Blue Yonder system with heterogeneous hardware.

In our scenario, the resource landscape consists of a heterogeneous hardware environment comprising two low-budget machines (`desc1` and `desc2`) and two high-end machines (`desc3` and `desc4`). Each of the low-budget machines is equipped with one AMD Athlon^{TM} Dual Core Processor 5200B with two cores and no hyper-threading. The high-end machines are equipped with an Intel® Core^{TM} i7-3770 CPU with four cores and hyper-threading, i.e., eight logical cores. All four hardware machines are located in the same network, connected with a 1 GBit Ethernet.

An example topology is depicted in Figure 8.16. In the depicted default setup, the database runs on a dedicated high-end machine. `GW` and `PS` instances can be distributed over the two low-budget machines and the second high-end machine.

8.4.1.1. Degrees of Freedom and Adaptation Process

The types of workload changes that occur in the system environment are changes in the customer workloads. A customer's workload is characterized by the service that is called (`train`, `predict`), the number of requests to the service (request arrival rate, typically one to ten requests at the same time), the execution type of the requests (sequential or parallel), and the requests' size (the number of records per request, typically varying between 10,000 and 500,000). To react on changes in the environment, additional `PS` instances can be started on other machines or they can be migrated between machines at run-time.

Service Type:	`train` vs. `predict`
Nr. of Requests:	1 to 10
Request Execution:	sequential vs. parallel
Record Size:	10,000 to 500,000 records per user requests
PS Deployment:	high-end vs. low-budget machines

Our experiments showed that the maximum number of `PS` instances per machine is limited by two times the available cores. If we try to *overcommit* a machine, i.e., deploy more `PS` instances than this limit, the performance decreases significantly due to resource contention. The challenge in this case study is that our approach is faced with a heterogeneous hardware environment and with different performance requirements of multiple concurrent customers. For example, upon a workload change of a given customer, the adaptation process has to decide whether to start/stop a `PS` on a low-budget or a high-end machine while taking into account the performance requirements and topology of other customers.

An adaptation of the system topology is triggered by an updated request schedule, i.e., a change in the workload of one or more customers. The goal of this triggered adaptation process is to find a deployment of `PS` instances such that the customer's SLAs are fulfilled and resources are used as efficiently as possible. The found deployment can then be transformed into a topology configuration that can be used to reconfigure the Blue Yonder system accordingly. Conceptually, the triggered adaptation process can be separated into the following steps: i) allocate new `PS` instances to find a deployment that fulfills all customer SLA's, ii) consolidate the deployment of the `PS` instances on less hardware resources, and iii) reduce `PS` instances as long as SLAs are fulfilled. Furthermore, as the process should be able to trade-off the requirements of different customers, there should exist an additional step that aims at resolving possible resource bottlenecks that may arise when customers share resources. For example, the allocation of an additional `PS` instance on a machine with `PS` instances of other customers might cause a performance degradation for the other customers due to resource contention.

8.4.2. Applied DML Instance

In this section, we present DML instance that describes the system architecture, performance behavior, and the adaptation process that adapts Blue Yonder's system to changing customer workloads. First, we introduce the architecture-level performance model of Blue Yonder's system that we use for performance analysis and then describe the modeled adaptation points and adaptation process.

8.4.2.1. Architecture-Level Performance Model

Figure 8.17 depicts in a UML-like notation a simplified version of the architecture-level performance model we created for the Blue Yonder system using DML. It shows the resource landscape, application architecture, usage profile, and deployment of Blue Yonder's system. The model also includes parametric descriptions of the performance behavior of the software components.

In the center of Figure 8.17, we see the resource landscape model, consisting of a DataCenter BYDC that contains the previously described hardware. The resource configuration specifications of the ComputingInfrastructures are attached as annotations. The ComputingInfrastructure nodes are connected with a 1 GBit Ethernet.

Figure 8.17 shows that there are four different component instances in the resource landscape: one gateway server instance (*GatewayServer*), two `PS` instances (*PredictionServerA* and *PredictionServerB*), and one database instance (*Database*). This depicted deployment is only one possible deployment variant of Blue Yonder's system. During an adaptation process, when the model is changed, the model instance may look different, e.g., further `PS` instances might be deployed on other machines.

Each of the depicted software components provides one or more of the previously described services. For some of these services, we have depicted the fine-grained description of their performance-relevant behavior. For example, the top of Figure 8.17 depicts the fine-grained behavior of the `predict` service offered by the *GatewayServer* component. This behavior contains two InternalActions that require a certain amount of CPU resources to parse and schedule the prediction job (parsePredictionJobs and schedulePredictionJobs, respectively). After parsing and scheduling the job, it is passed to one of the available `PS` instances. In this example of the service's behavior descriptions, the probability for each branch is 50% as there are only two `PS` instances. However, during the adaptation process, when the number of `PS` instances changes, the respective probabilities have to be adjusted accordingly.

Figure 8.17 also shows an example for the usage profile model. In the depicted example, ten customers use the `train` service with a record size of 500,000.

In contrast to the case study presented in Section 8.2, we were not able to automatically extract an architecture-level performance model of the Blue Yonder system due to the lack of detailed monitoring tools. Thus, we created the model instance manually. To obtain a representative performance model, we conducted a large set of experiments varying different parameters to analyze the dependencies of the resource demands on these parameters (Schott, 2013; Rattu, 2012). The parameters we varied in our experiments were mainly induced by the parameters that vary at run-time: the requested service type (`train` vs. `predict`), the number of requests (request arrival rate), the record size (10,000 to 500,000 records per user request), the execution type of the requests (sequential or parallel), and the number of parallel requests (1 to 10). Furthermore, to investigate the impact of the heterogeneous hardware environment and the mutual influences of multiple `PS` instances, we also varied the `PS` deployment (high-end vs. low-budget machine) and the number of `PS` instances (1 to 8).

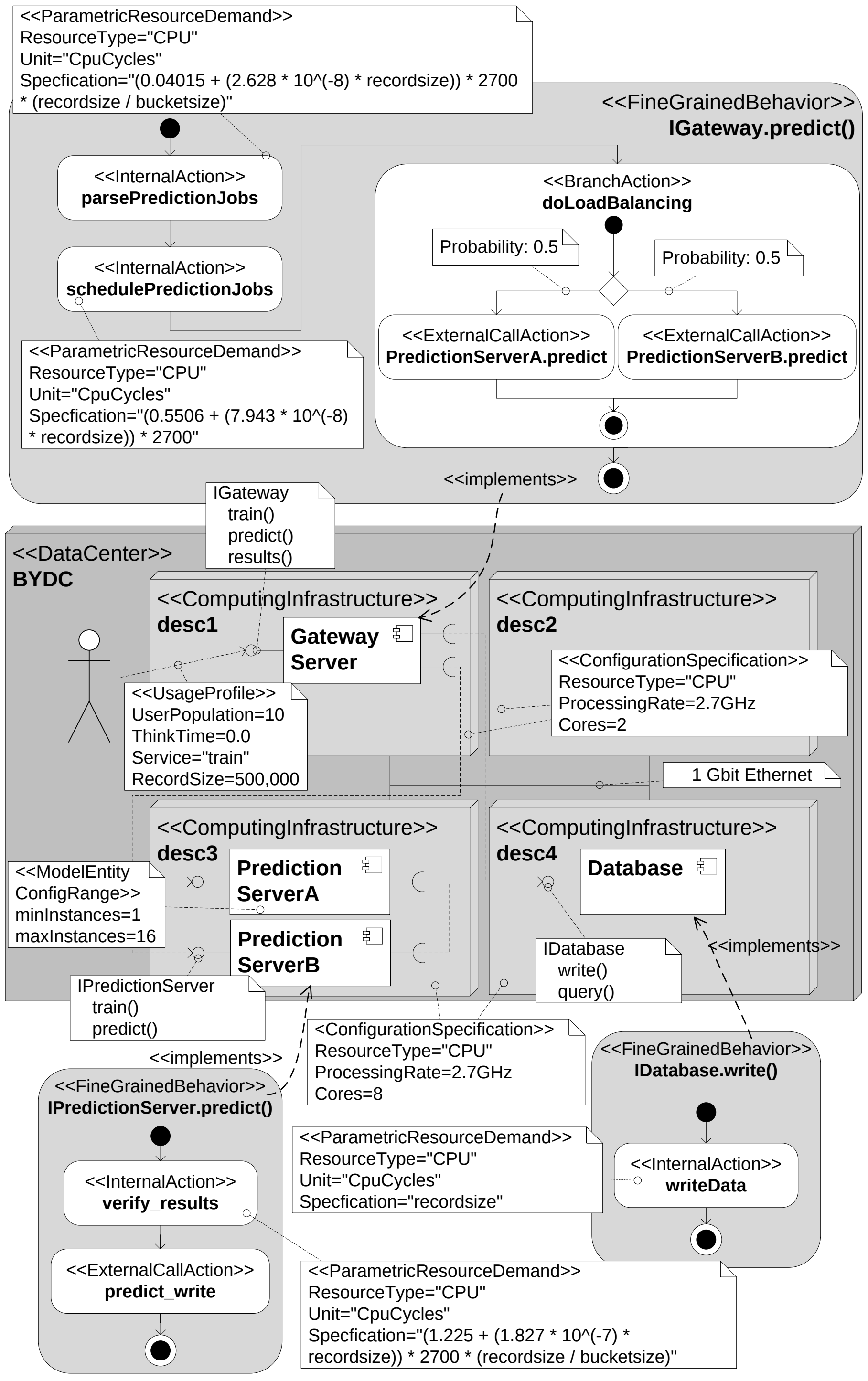


Figure 8.17.: DML instance describing a deployment of the Blue Yonder system.

The metrics we observed during our experiments to derive the service resource demands were average CPU utilization and average response time, i.e., the time a user request spends in the system. To obtain these data, we measured the request throughput and the CPU utilization during the experiments and extracted the timing values from the log files of the system. We then used the R framework for statistical computing (R Project, 2013) to derive the resource demands for the modeled InternalActions using linear regression. For example, the parametric resource demand *rd* derived for `schedulePredictionJobs` depicted in Figure 8.17 is

$$rd = (0.5506 + (7.943 \cdot 10^{-8} \cdot \texttt{recordsize})) \cdot 2700$$

This resource demand depends on an external influencing parameter `recordsize` that corresponds to the number of records in the user request. Note that the additional multiplication by 2700 is necessary to adjust the resource demand to the processing rates of the hardware (cf. Figure 8.17). We have derived such parametric resource demands for all InternalActions in our model and more details can be found in the master's theses of Schott (2013) and Rattu (2012).

To evaluate the accuracy of the performance predictions provided by the model, we conducted several experiments that are independent of the experiments we used to derive the service resource demands. In Figure 8.18, we compare the predicted with the measured response times of the `train` and `predict` services for five parallel user requests with a varying amount of `PS` instances. The figure shows that the response time of the `train` service improves until five `PS` instances are used, whereas the `predict` response time improves further. The reason is that the five parallel `train` requests in this experiment can be distributed to five `PS` instances, i.e., an additional `PS` instance does not speed-up the training. In contrast, `predict` requests can be split into several jobs which can be distributed over an arbitrary number of `PS` instances to speed-up performance. This experiment result confirms that the modeled behavior of the Blue Yonder system is representative.

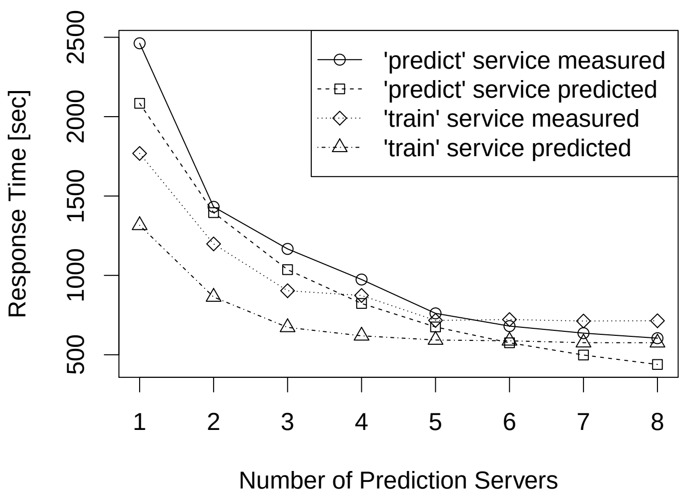


Figure 8.18.: Comparison of predicted and measured response times of the `train` and `predict` services for five parallel requests and a varying number of `PS` instances running on the high-end machine `desc4`.

In another scenario we evaluated the prediction accuracy for different workload mixes. Table 8.6 shows the absolute and relative prediction error for the average response time and Table 8.7 shows the absolute prediction errors for the CPU utilization on different hardware nodes. The results showed that the prediction error did not exceed 30%.

Table 8.6.: Measured and predicted average response times and their relative errors for nine parallel `predict` requests for varying mixed data record sizes with six `PS` instances allocated on `desc4`.

Record Sizes	**Response Time [sec]**		**Error**
[in 1,000 records]	measured	predicted	[in %]
50 & 100	194.50	175.26	-9.9
100 & 200	366.48	325.16	-11.3
150 & 300	545.05	485.65	-10.9
250 & 500	937.24	780.91	-16.7

We also conducted further experiments to evaluate the accuracy in situations where multiple customers use the system in parallel or where we vary the amount and deployment of the `PS` instances (Schott, 2013). The results also confirmed the prediction error for response time and utilization of approximately 30%.

Table 8.7.: Measured and predicted average CPU utilization and their absolute errors for nine parallel `predict` requests for varying mixed data record sizes with six `PS` instances allocated on `desc4`.

Record Sizes	`desc2` [%]			`desc3` [%]			`desc4` [%]		
[in 1,000 records]	meas.	pred.	err.	meas.	pred.	err.	meas.	pred.	err.
50 & 150	9.42	27.6	18.2	17.11	6.0	11.1	51.56	38.9	12.7
100 & 200	10.35	19.4	9.1	16.54	4.0	12.5	49.19	40.8	8.4
150 & 300	10.51	16.2	5.7	16.33	3.3	13.0	47.79	41.4	6.4
250 & 500	10.20	13.7	3.5	16.65	2.8	13.9	44.67	41.9	2.8

8.4.2.2. Adaptation Points Model and Adaptation Process Model

To apply our model-based performance-aware resource management approach, we first have to define and model the adaptation points of the system we want to adapt. Based on the adaptation points, we can then model the adaptation process.

In our case study, we consider multiple adaptation points for the Blue Yonder system. The adaptation points are customer-specific, i.e., they exist for each customer's `PS` instances deployed in the system. First, we can increase or decrease the number of `PS` instances that are assigned to a customer. The minimal number of `PS` instances is one. The maximum number of `PS` instances is limited to two times the available cores of the machine. The reason is that our experiments indicated a significant performance degradation due to resource contention if we increase the number of `PS` Instances above this limit.

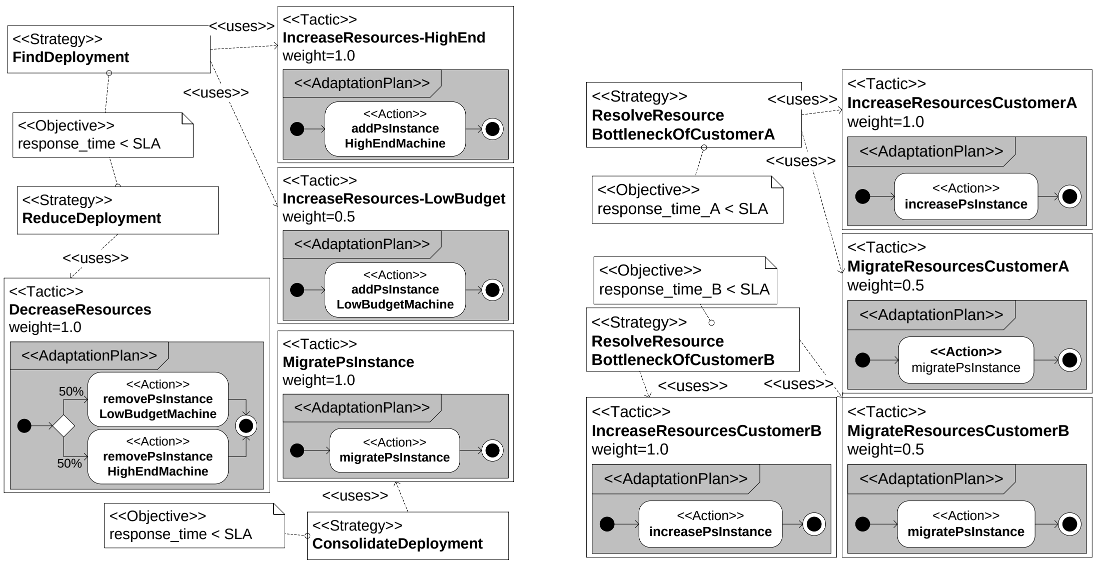
The second adaptation point is the deployment of `PS` instances. By starting and stopping `PS` instances or by consolidating `PS` instances on fewer machines, we can improve resource efficiency and lower operational costs. For example, it can be beneficial to consolidate the `PS` instances of multiple low-budget machines on a single high-end machine.

Finally, the number of `PS` instances in a data center is also limited to avoid significant performance degradation due to resource contention. The adaptation points can be specified as follows with S being the set of servers of the Blue Yonder environment that do not host the database:

$$\begin{aligned}
\textit{Number } \texttt{PS} \textit{ Instances per server } s\text{:} \quad & \text{Min: } 1, \quad \text{Max: } \#cores(s) \cdot 2 \\
\textit{Deployment of } \texttt{PS} \textit{ Instances}\text{:} \quad & S \\
\textit{Number of } \texttt{PS} \textit{ Instances per Data Center}\text{:} \quad & \text{Min: } 1, \quad \text{Max: } \textstyle\sum_{s \in S} \#cores(s) \cdot 2
\end{aligned}$$

In the adaptation points model, these numbers are specified as OCL constraints that can be checked on the architecture-level performance model.

As mentioned previously, the trigger of an adaptation is an updated *request schedule*. The request schedule specifies which customers will request which services and with what amount of data. Furthermore, it also contains the customer-specific SLAs. The purpose of the adaptation process presented in the following is to allocate available resources among Blue Yonder's customers such that their SLAs are fulfilled. Moreover, the resources should be used as efficiently as possible to save operating costs. Thus, the output of our adaptation process is a deployment model that fulfills the required SLAs. The found deployment model can then be transformed into a topology configuration that can be used to reconfigure the Blue Yonder system accordingly. In the following, we describe an adaptation process that fulfills these requirements. It is modeled using the adaptation process modeling language presented in Section 6.2. To execute the adaptation process, we use our adaptation framework presented in Section 6.3. It interprets the adaptation process model and adapts the previously introduced architecture-level performance model accordingly.


<<Strategy>> FindDeployment
<<uses>>
<<Tactic>> IncreaseResources-HighEnd weight=1.0
<<AdaptationPlan>>
<<Action>> addPsInstance HighEndMachine
<<uses>>
<<Objective>> response_time < SLA
<<Strategy>> ReduceDeployment
<<Tactic>> IncreaseResources-LowBudget weight=0.5
<<AdaptationPlan>>
<<Action>> addPsInstance LowBudgetMachine
<<uses>>
<<Tactic>> DecreaseResources weight=1.0
<<AdaptationPlan>>
50%
<<Action>> removePsInstance LowBudgetMachine
<<Action>> removePsInstance HighEndMachine
50%
<<Tactic>> MigratePsInstance weight=1.0
<<AdaptationPlan>>
<<Action>> migratePsInstance
<<uses>>
<<Objective>> response_time < SLA
<<Strategy>> ConsolidateDeployment
<<Strategy>> ResolveResource BottleneckOfCustomerA
<<uses>>
<<Tactic>> IncreaseResourcesCustomerA weight=1.0
<<AdaptationPlan>>
<<Action>> increasePsInstance
<<uses>>
<<Objective>> response_time_A < SLA
<<Objective>> response_time_B < SLA
<<Tactic>> MigrateResourcesCustomerA weight=0.5
<<AdaptationPlan>>
<<Action>> migratePsInstance
<<Strategy>> ResolveResource BottleneckOfCustomerB
<<uses>>
<<uses>>
<<Tactic>> IncreaseResourcesCustomerB weight=1.0
<<AdaptationPlan>>
<<Action>> increasePsInstance
<<Tactic>> MigrateResourcesCustomerB weight=0.5
<<AdaptationPlan>>
<<Action>> migratePsInstance

(a) *FindDeployment*, *ReduceDeployment*, and *ConsolidateDeployment*

(b) *ResolveResourceBottleneck* for two customers A and B

Figure 8.19.: Schematic representation of the adaptation process model used in this case study.

The adaptation process that we use in this case study consists of the following strategies: *FindDeployment*, *ReduceDeployment*, and *ConsolidateDeployment*. The *FindDeployment* strategy launches new PS instances until all customer SLAs are fulfilled. It contains two different tactics starting the new PS instances on low-budget machines and high-end machines, respectively. *ReduceDeployment* removes unnecessary PS instances from machines to save operating costs, e.g., if the workload of a customer has decreased. Finally, *ConsolidateDeployment* migrates PS instances between machines with the goal to improve efficiency. A schematic representation of the adaptation process model is depicted in Figure 8.19a for one customer. All these strategies exist for each customer that has PS instances deployed in this environment. Furthermore, as the process should be able to trade-off the requirements of different customers, we have also defined an additional strategy *ResolveResourceBottleneck* that aims at resolving possible resource bottlenecks that may arise when customers share resources. Figure 8.19b depicts a schematic representation of these strategies for two example customers A and B.

8.4.3. Evaluation

Blue Yonder currently uses dedicated resources for each customer to fulfill their respective SLAs. The amount of resources is estimated based on the experience of Blue Yonder and can range from few low-budget desktop machines to hundreds of cores on high-end servers. Consequently, Blue Yonder is interested in dynamically adapting the amount of resources according to the actual customer demand. The following scenarios specifically address the evaluation questions Q1 and Q4. We evaluate the applicability of our approach in adapting Blue Yonder's heterogeneous resource environment to changes in the customer workloads such that the customer performance requirements are satisfied while ensuring efficient resource usage (Q1). We show that the approach is applicable in a heterogeneous resource environment and that it can trade-off diverging performance requirements of different customers (Q4).

8.4.3.1. Scenario 1: Adjusting Resources to Workload Changes Considering a Heterogeneous Resource Environment

The goal of this scenario is to evaluate the effectiveness of our approach in adapting resource allocations to workload changes such that customer SLAs are fulfilled while considering the heterogeneous nature of the hardware environment.

Our scenario starts with the default topology (one `GW` on `desc2`, one `PS` on `desc4`, one `DB` on `desc3`) and a customer that issues one `predict` request with 500,000 records. We assume that all records of this customer must be completed within 3,600 seconds (one hour). The default topology is sufficient to process this load without SLA violations.

However, if the customer increases the number of requests to 50 (indicated by the dotted line in Figure 8.20), the default topology is not able to handle the load within the given time. Thus, this workload increase event triggers the `FindDeployment` strategy with the objective to find a system configuration that can handle the load without SLA violations. Since the `IncreaseResources-HighEnd` tactic initially has the highest weight, the adaptation framework selects this tactic to add resources to the system. By applying the tactic, another `PS` instance is launched on the high-end machine `desc4`, which improves the response time of the system towards the strategy's objective, but SLAs are still violated. Thus, the adaptation framework continues applying this tactic.

After the 16th iteration, the response time cannot be further improved. This corresponds to our experience from the experiments that the maximum number of `PS` instances on each machine is limited by two times the available cores. If we try to overcommit the resources of a machine, i.e., deploy more `PS` instances per machine than this limit, the performance decreases significantly due to resource contention. For example, for the low-budget machines where we have two cores without hyper-threading, four `PS` instances can be executed in parallel with negligible resource contention. Given that after 16 iterations, applying the tactic did not further improve the response time of the system, the adaptation framework revokes the application of this tactic and decreases its weight. However, since the objective has not been achieved yet, the adaptation process continues and applies the `IncreaseResources-LowBudget` tactic in the next iteration. This tactic adds another `PS` instance to the low-budget machine `desc1`, which further improves the response time. The adaptation framework keeps applying this tactic for three additional iterations until all SLA violations are resolved. In summary, our modeled adaptation process suggests a deployment of 20 `PS` instances, four on `desc1` and 16 on `desc4`. As Figure 8.20 shows, this deployment is sufficient to handle the increased load. This experiment shows that with our approach we are able to model an adaptation process that utilizes the available capacity without overcommitting the resources.

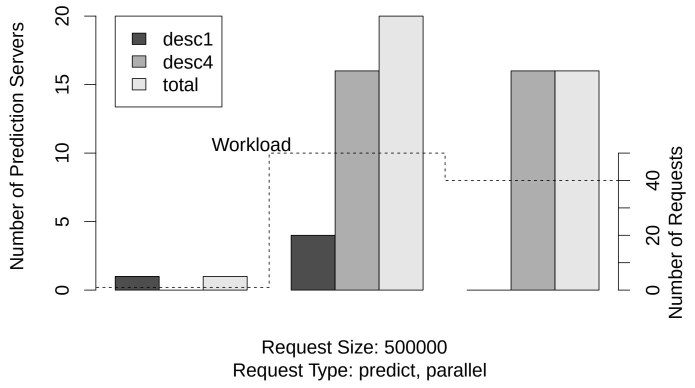


Figure 8.20.: Amount of `PS` instances in the system after adapting the system environment to changes in the workload of a customer.

In the next step, we reduce the workload from 50 to 40 parallel requests, i.e, less resources should be sufficient to maintain the SLA of one hour. Thus, the `ReduceDeployment` strategy is triggered. The adaptation framework applies the `DecreaseResources` tactic to reduce the amount of active `PS` instances. This tactic is repeated until SLAs start being violated, indicating that the process has removed too many `PS` instances. At that point, the application of the latest iteration of the tactic is revoked, and the adaptation process ends. However, the system configuration after applying this strategy may be further optimized in case `PS` instances are distributed over several machines that can be consolidated. To this end, the `ConsolidateDeployment` strategy applies the `MigrateResources` tactic, migrating the `PS` instances from the low-budget machine to the high-end server. As a result of our adaptation process, the four `PS` instances running on the low-budget machine are released (cf. Figure 8.20).

8.4.3.2. Scenario 2: Trading-Off Resource Allocations Between Customers

In this scenario, we show that our approach is applicable in scenarios where changes of the workload behavior of one customer affect the performance experienced by other customers. The goal is to show that our approach, compared to trigger-based approaches, can trade-off different performance requirements of customers with different priorities. Therefore, we have added two further strategies to our adaptation process model depicted in Figure 8.19b.

The initial Blue Yonder topology in this scenario comprises four `PS` instances that are deployed on `desc1` (see Figure 8.21). Two of these `PS` instances belong to customer A, which is a gold customer. The other two `PS` instances belong to customer B and C, respectively, which are silver customers. A gold customer is a customer with higher priority, i.e., violating this SLA causes higher penalties. Thus, to minimize penalties caused by major response time fluctuations, gold customers have the additional constraint that `PS` instances of such customers must not be executed on machines that are overcommitted. Overcommitted machines execute two times more `PS` instances than there are available cores (cf. previous scenario).

In this scenario, we assume that we observe an SLA violation for customer B due to a workload increase (cf. Figure 8.22), triggering our adaptation process. The process first applies

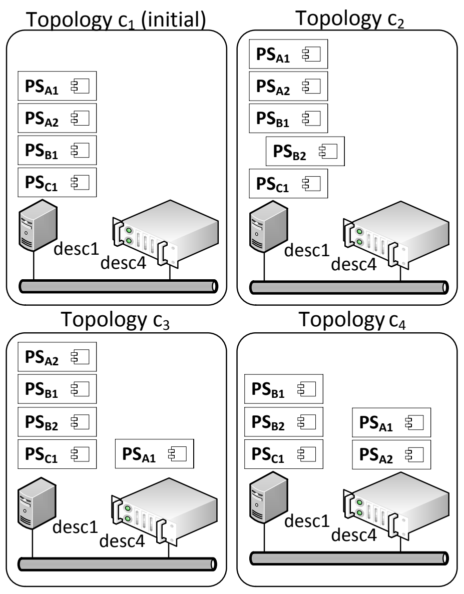


Figure 8.21.: Details of the different system configurations explored as part of the adaptation process triggered by a workload change of one customer affecting other customers.

the `IncreaseResourcesCustomerB` tactic of the `ResolveResourceBottleneck-OfCustomerB` strategy, because this tactic has the highest weight. Applying this tactic starts another `PS` instance for customer B on `desc1` (topology c_2). However, in the new topology, the SLAs of the gold customer A are now violated due to mutual performance influences. This SLA violation triggers the `ResolveResourceBottleneckOf-CustomerA` strategy and the adaptation framework executes the `IncreaseResources-CustomerA` tactic. However, this tactic cannot be applied because of the constraint that `PS` instances of gold customers must not be executed on overcommited machines. As a result, the tactic is revoked and its weight is reduced. In the next iteration, the adaptation process applies the `MigrateResourcesCustomerA` tactic to migrate a `PS` instance of customer A to `desc4` (topology c_3). The migration reduces response times, but still does not eliminate the SLA violation of customer A. Nevertheless, the tactic contributed towards the strategy's objective, and thus, the adaptation process continues by migrating the second `PS` instance of customer A to `desc4` (topology c_4). This resolves the problem and the adaptation process completes.

This scenario shows that using our model-based approach we can take into account the mutual performance influences between different customers (adding a new `PS` instance for customer B affected customer A) and model a process that ensures SLA compliance for all customers. In the considered scenario, a conventional trigger-based approach would simply add a `PS` instance. The issue that adding this further instance leads to an SLA violation for customer A would only be detected after the system has been reconfigured. Of course, then a new trigger would start further adaptations to address this issue, however, penalty costs would arise due to the SLA violations that will already have occurred.

8.5. Discussion

The presented case studies are illustrative and representative examples of how our model-based approach can be used to implement performance and resource management at the system architecture level. In different scenarios, we showed how to use the modeling

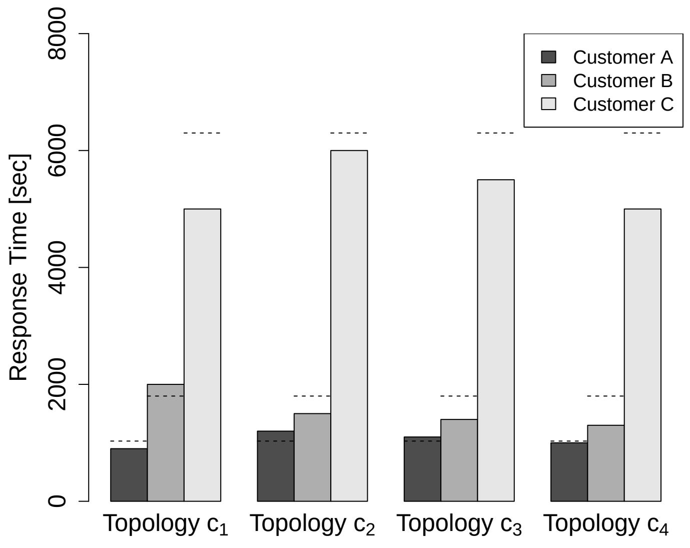


Figure 8.22.: Response times of the different customers during the adaptation process (SLAs are denoted by the dashed lines).

abstractions presented in this thesis to specify adaptation processes that leverage architectural and performance-relevant information about the system for automated adaptation decisions. More specifically, the results demonstrate that: i) our approach can be used to describe the performance-relevant aspects of homogeneous and virtualized resource environments as well as of heterogeneous resource environments consisting of low-budget desktop machines and high-end servers, ii) the model-based approach can be applied in different domains such as business information systems (SPECjEnterprise2010) and compute-intensive applications (Blue Yonder example application), iii) the adaptation process meta-model can be used to describe different types of adaptation processes, and iv) our approach can be effectively used to adapt a system proactively, i.e., before performance or resource efficiency problems occur. In the following, we discuss the overall results obtained in these case studies with respect to the overhead and efficiency of the approach, and discuss possible threats to validity.

In the presented case studies, we focused on analyzing the performance of our approach compared to reactive, trigger-based adaptation approaches. The results in the considered scenarios showed that both the reactive and the proactive adaptation approach needed only approximately 45% of the resources of the static allocation approach. Furthermore, the proactive approach reduced SLA violations by up to 60%. Moreover, our approach enables the search for a solution to an anticipated or detected problem at the model level without having to experiment with the actual system. Thereby, it is possible to save costly adaptations on the real system.

If we analyze the overhead of our proactive model-based system adaptation in more detail, we can divide it into the factors depicted in Figure 8.23: the overhead for workload classification and forecasting and the overhead of the model-based adaptation process. Regarding the overhead of our workload classification and forecasting approach, our experiments presented in Section 7.4 with WCF on a single-core machine showed that the overhead ranges within seconds up to a few minutes depending on the data and configuration settings. Compared to the overhead for workload classification and forecasting, the overhead for model-based system adaptation is significantly higher. This overhead is determined by two main factors: the number of iterations to find a solution at the model level and the overhead caused by the performance analysis of the model for each iteration. The number

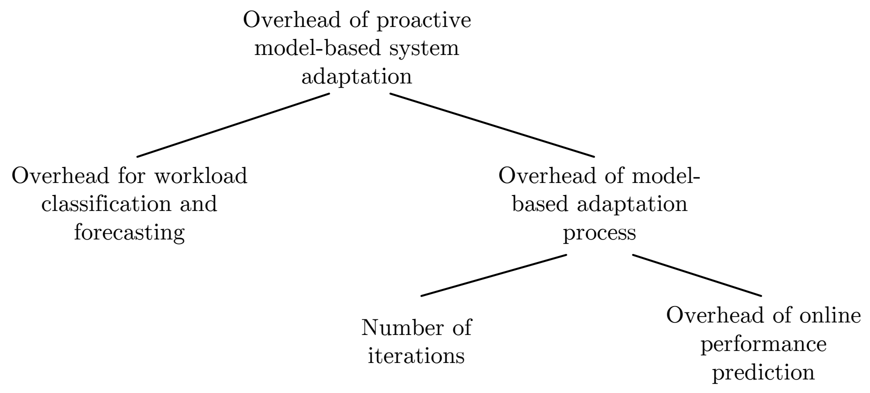

Overhead of proactive model-based system adaptation
Overhead for workload classification and forecasting
Overhead of model-based adaptation process
Number of iterations
Overhead of online performance prediction

Figure 8.23.: Major overhead factors of proactive model-based system adaptation.

of iterations to find a solution depends on the specification of the adaptation process. The more efficient the design of the adaptation process with respect to the adaptation goals, the less iterations are required to find a solution. The overhead for performance model analysis depends on the analysis techniques used for performance prediction and the complexity of the model. In the multitude of experiments conducted for this thesis as well as in the thesis of Brosig (2014), the time to obtain results varied between seconds and a few minutes in the worst case. In summary, the adaptation overhead of our approach is dominated by the number of iterations to find a suitable model configuration multiplied with the time for performance model analysis.

Thus, we have to discuss whether the model-based approach is quick enough to provide a solution before actual performance or resource efficiency problems occur. The results of the first case study presented in Section 8.2 using SPECjEnterprise2010 as a representative business information system show that the 15 minute time window in which we receive updates of the request arrival rates is short but sufficient to find a solution and adjust the system accordingly. In the case study with Blue Yonder's compute-intensive application presented in Section 8.4, the execution of requests can take up to several hours and thus, the search for a solution with our model-based approach is even less critical since its overhead is negligible compared to the typical response time of a request. Finally, the case study presented in Section 8.3 shows that WCF can be effectively applied at run-time to provide workload forecasts within the given time constraints.

These considerations show that the overhead of our approach is justifiable and the results demonstrate that the approach can deliver suitable solutions in time in all investigated scenarios.

With our end-to-end validation, we comprehensively demonstrated the applicability of our model-based approach for proactive performance-aware resource management in different systems and configurations. Nevertheless, typically asked questions threatening the validity of our results are the question how representative the presented case studies are and how realistic our settings are compared compared to the settings in real-life data centers. To reduce this threat and to increase the external validity of our validation as far as possible, we used our available resources to create setups that are as realistic as possible without making system-specific assumptions, and selected evaluation scenarios based on real-life problems, e.g., of our industrial partner Blue Yonder. To create realistic setups, in our infrastructure we employed technologies that are extensively used in industry (Xen and VMware hypervisors, Oracle Weblogic Application Servers, MySQL and Oracle Database Systems). As software services hosted on the infrastructure, we used real-life applications from industry (Blue Yonder example project) or applications that are recognized by industry as representative standard benchmarks (SPECjEnterprise2010). As these applications belong to different domains—business information systems (SPECjEnterprise2010) and compute-intensive applications (Blue Yonder)—our case studies also cover different situations with different requirements and needs. Finally, we also used different realistic workloads obtained from the Wikimedia Project (2011) and IBM. For these reasons, we are convinced of the representativeness of the case studies and the obtained results.

A further crucial question is whether our approach is generally applicable for proactive system adaptation. The answer is that proactive adaptations can be performed only in situations where the workload of the adapted system can be forecast with sufficient accuracy. This limitation of our approach comes from the limitations of the forecasting methods we employ in our Workload Classification and Forecasting (WCF) approach. As these methods normally work well for workloads that exhibit a certain degree of seasonal or trend behavior, WCF and therefore our proactive system adaptation is limited to applications with such a workload intensity behavior. Thus, in situations with a completely arbitrary workload intensity behavior, it is not possible to apply WCF reliably. Nevertheless, our model-based adaptation approach can still be applied in a reactive manner.

Finally, there remains the question of whether the comparison of our approach with other adaptation approaches is sufficient. The preferred way to compare our approach with other solutions in this area would be to use a benchmark for self-adaptive systems. However, currently such benchmarks do not exist (Almeida and Vieira, 2011). A second option would be to apply related approaches (such as the ones mentioned in Section 3.1.2) in the context of our case studies. The problem is that these approaches are either targeted at other QoS aspects besides performance or they focus only on adaptation at the application level and do not include the system's operational environment as part of the considered adaptation possibilities. However, for performance and particularly for resource management, it is inevitable that the resource environment is considered explicitly as part of the adaptation process and the underlying system models. Furthermore, it is also technically unfeasible to apply alternative research approaches due to deviating underlying assumptions, such as the employed performance or QoS models, and the lack of publicly available tools or unsupported features. For these reasons, we focused on comparing our approach with static resource allocation and reactive, trigger-based approaches, which are predominantly used in industry today. Although we have not directly deployed our case study applications in a real-life threshold-based system such as, e.g., the Amazon Elastic Compute Cloud (Amazon Web Services, 2010), the behavior of our reactive approach and such systems is closely related. Nevertheless, due to the careful selection of the case studies and the various considered application scenarios we are convinced of the general applicability of our approach and the external validity of the results.

9. Conclusions and Outlook

In this chapter, we conclude this thesis with a summary of the contributions presented in this thesis. Moreover, we discuss ongoing and future work in the area of engineering self-adaptive software systems and autonomic performance-aware resource management.

9.1. Summary

Recent trends like cloud computing and virtualization show that service providers are adopting to more flexible and dynamic systems. The motivation for this trend is that service providers want to allocate resources at run-time depending on changes in the system environment to reduce operating costs. However, due to the inherent complexity of dynamic systems, the development of methods that automatically adapt a system to changes in its environment is a challenging task (Kramer and Magee, 2007; Blair et al., 2009). In Chapter 1, we argued that sophisticated modeling techniques are needed, specifically designed for performance and resource management at run-time. These techniques should allow capturing both static and dynamic aspects of the managed system, including all performance-relevant influences of the system's execution environment, its architecture, as well as its adaptation space, adaptation strategies and processes, in a generic, human-understandable and reusable way. Moreover, model-based system adaptation mechanisms are needed that apply such modeling and prediction techniques end-to-end to drive autonomic decision-making at run-time. In contrast to trial-and-error approaches working on the system directly, with model-based system adaptation mechanisms it is possible to search for suitable system configurations at the model level and thereby avoid unnecessary and possibly costly adaptations.

In this thesis, we presented a systematic approach for engineering self-adaptive systems with autonomic performance-aware resource management capabilities. Core concepts of this approach is a **process model for proactive model-based system adaptation** which is based on the Descartes Modeling Language (DML), a novel architecture-level modeling language specifically designed to support the autonomic adaptation process by leveraging online performance prediction and proactive model-based system adaptation techniques (cf. Chapter 4). In summary, DML provides sophisticated modeling abstractions to describe two major concerns of autonomic performance-aware resource management at run-time. First, it provides modeling abstractions to create performance models of the system that reflect the performance-relevant properties and behavior of the system and its distributed infrastructure at the architecture-level. Such architecture-level performance models serve as the basis for online performance prediction techniques which can be used to continuously predict at run-time a) the effect of changes in the system environment, such as application workloads, on the system performance and resource efficiency, and b) the impact of possible adaptation actions. Second, DML provides generic and flexible formalisms to model the dynamic aspects of self-adaptive systems and their adaptation processes. This abstraction from technical and system specific details reduces the inherent complexity of modern dynamic IT infrastructures and services and thereby eases the systematic development of self-adaptive systems.

In Huber et al. (2011a), we showed that architecture-level performance models provide a suitable abstraction layer for model-based performance and resource management. However, current architecture-level performance models are typically targeted at system design-time analysis and abstract from the complexity of modern dynamic IT service infrastructures. In Chapter 5, we introduced **modeling abstractions to describe the structure and configuration** (e.g., memory, bandwidth, and so on) of both physical and logical resources of modern distributed IT infrastructure. Innovative aspects of these modeling abstractions are that they provide constructs to model the hierarchy of nested resource layers as well as the influences of these layers on the overall system performance. We showed that considering the performance influences of nested resource layers, e.g., the overhead introduced by virtualization, is important for obtaining detailed performance predictions. Furthermore, we showed that structural details of the resource landscape are important to be considered when specifying adaptation processes as well as for decision-making during the adaptation process. The proposed modeling abstractions have been published in Huber et al. (2012a).

To ease the quantification of the performance-influencing properties of logical resource layers (such as virtualization or middleware), we also presented a **method for the identification, classification, and automatic quantification of performance-influencing properties**, using virtualization as proof-of-concept (cf. Section 5.2). This method and the results obtained for the XenServer virtualization platform have been published in Huber et al. (2010b). The evaluation of the results on VMware ESX as well as the derivation of a performance model to estimate the overhead of virtualization platforms has been published in Huber et al. (2011b).

In Chapter 6, we proposed **modeling abstractions to describe the degrees of freedom** of the system that can be employed for run-time system adaptation. Using the presented architecture-level performance model as a basis, these modeling abstractions simultaneously describe the valid configuration space of the system. We also proposed a **flexible modeling language to specify adaptation processes** that keep the system in a state such that its operational goals are continuously fulfilled. With this modeling language, it is possible to specify adaptation processes at the model level in a generic, human-understandable and reusable way. To execute the specified adaptation processes, we presented a framework that interprets and executes the modeled adaptation processes on the architecture-level performance model. The framework relies on the online performance prediction techniques by Brosig (2014) to assess the impact of the adaptation and consider this during the adaptation process accordingly. With this approach, it is possible to abstract from technical, system-specific details and shift the inherent complexity of system adaptation to the adaptation framework. The modeling formalism to describe the system adaptation possibilities has been published in Huber et al. (2012a). The concepts of the proposed modeling language for specifying adaptation processes has been initially published in Huber et al. (2012b). An extension of these concepts as well as the architecture of the adaptation framework have been presented in Huber et al. (2014).

To proactively adapt the system, we presented a method for **self-adaptive workload classification and forecasting at run-time** (cf. Chapter 7). This method automatically identifies relevant characteristics of given workloads and selects suitable forecasting methods according to the configured user-specific forecasting objectives. The dynamic design of this method and the flexibility to continuously adapt the selection of the forecasting method based on the observed forecasting accuracy increases the overall accuracy of the forecast results. Our experiments showed that the proposed method supports continuous forecasts at run-time with significant accuracy improvements and controllable computational overheads. The results of the workload forecasts provide the input for online performance prediction with the goal to predict the impact of the forecast workload intensity change on the system performance and to trigger subsequent system adaptation if necessary. The proposed method and the achieved results have been published in Herbst et al. (2013a) and refined in Herbst et al. (2014).

Finally, we integrated all individual parts to a coherent approach for autonomic performance-aware resource management, implementing the presented concepts of a holistic model-based adaptation control loop. In Chapter 8, we presented the **end-to-end validation of our approach in three different representative case studies**, demonstrating that our model-based approach can be effectively used for autonomic performance-aware resource management. The selected evaluation scenarios have been derived from typical real-life problems of, e.g., our industrial partner Blue Yonder, a leading service provider in the field of predictive analytics and big data. With the presented case studies, we covered a broad spectrum of configurations with different types of applications, hardware environments, deployment configurations, and workloads to investigate the applicability of our approach under various conditions. For example, as applications we selected the SPECjEnterprise2010 benchmark application modeling a representative, state-of-the-art business information system, and a real-world compute-intensive application from Blue Yonder. The employed hardware environments consisted of homogeneous virtualized cluster environments as well as heterogeneous environments consisting of low-budget and high-performance machines. As workloads, we used different real-life workload traces from Wikimedia Project (2011) and our industrial collaboration partners.

In summary, the validation results showed that our approach can provide significant resource efficiency gains of more than 50% without sacrificing performance guarantees, and that it is able to trade-off performance requirements of different customers in heterogeneous hardware environments. Furthermore, we showed how our approach enables proactive system adaptation, reducing the amount of SLA violations by 60% compared to trigger-based approaches. From the scientific perspective, the results of these case studies show that DML allows the specification and execution of proactive system adaptation processes at the model level to perform autonomic system adaptation such that the system's operational goals are maintained. This shows that architecture-level performance models and online performance prediction can be effectively used for proactive autonomic performance-aware resource management, thereby achieving significant performance and resource efficiency gains in modern dynamic IT service infrastructures.

9.2. Outlook

The approach and results presented in this thesis provide the basis for several opportunities for ongoing and future work, summarized in the following paragraphs.

Modeling Additional QoS Properties

The current version of the Descartes Modeling Language (DML) is focused on performance including capacity, responsiveness and resource efficiency aspects. In future work, it is desirable to extend DML abstractions to support modeling additional QoS properties. For example, the architecture-level performance model could be extended to support mean time to failure annotations, such that it is possible to reason about the reliability of the system. This information could also be used to specify adaptation processes for increasing the system availability and reliability, e.g., by replicating critical VMs or software components. Furthermore, information about additional QoS properties could also be used to trade-off different QoS requirements during the system adaptation process. Thereby, DML could also be used for engineering more dependable and secure systems.

Refining Network and Storage Infrastructure Models

Currently, the modeling abstractions of the resource landscape model of DML are focused on computational resources and provide only basic possibilities to model networking or storage infrastructure resources. However, other researchers are already working on extending DML to provide sophisticated modeling abstractions for the network and storage infrastructure (Rygielski et al., 2013a,b; Noorshams et al., 2013c,b). In the future, these extensions should be integrated into DML to support online performance prediction for these infrastructure types. Thereby, it would be possible to better estimate the impact of these resource types on the performance and resource efficiency of the system and consider this during the system adaptation process.

Explicit Consideration of Adaptation Costs

During an adaptation process, different adaptation actions might exhibit different costs in terms of execution time or impact on the performance and efficiency of the running system. For example, a VM migration takes more time than adding virtual resources and has a significant impact on the network performance. On the other hand, the performance gain of VM migrating could be higher than adding virtual resources. Thus, it would be interesting to investigate methods to quantify the adaptation cost of different adaptation actions and to extend the modeling abstractions to express such costs explicitly. Then, the expressed costs can be considered in the adaptation process to trade-off adaptation costs with their achieved impact on system performance and efficiency.

Empirical Validation

To quantify the extent to which our model-based approach eases the development process of self-adaptive systems, it would be interesting to conduct empirical studies in large-scale data centers. In these studies, our approach would be applied from scratch, and system administrators unexperienced with DML would define system adaptation processes using DML as modeling formalism. These studies could help to assess how our model-based approach helps to reduce the complexity engineering self-adaptive software systems. Furthermore, they could also reveal possible improvements of the approach and pointers for future work.

Benchmarks for Self-Adaptive Systems

As mentioned in Section 8.1, it is a challenging task to develop benchmarks for self-adaptive software systems (Almeida and Vieira, 2011) and to develop metrics that can be used to compare different approaches. The case studies with their different evaluation scenarios and goals that have been presented in this thesis provide a good starting point for designing and developing benchmarks for self-adaptive software systems. Thus, future work could elaborate suitable ways to distribute the case studies or search for ways to provide the whole infrastructure setup such that other researchers and practitioners could compare related approaches or test new adaptation processes. The case studies could also lend themselves very well to evaluate the set of metrics that has recently been published by Herbst et al. (2013b) to quantify the elasticity of self-adaptive systems.

Self-Aware Computing Systems

The long-term vision of the Descartes Research Project—the research project that funded this thesis—is to develop a new method for the engineering of self-aware computing systems. Such systems are designed from the ground up with built-in online QoS prediction and self-adaptation capabilities used to enforce QoS requirements in a cost- and energy-efficient manner (cf. Section 2.1.4). The concepts presented in this thesis as well as in the thesis of Brosig (2014) lay the foundation for this vision. In the next steps, the presented model extraction, prediction, and adaptation techniques must be seamlessly integrated and tightly coupled to the system. In the future, the overall approach should be applied in industrial cooperations to showcase the applicability of our approach and thereby establish the vision a self-aware computing paradigm where computing systems are designed from the ground up with built-in self-reflective, self-predictive, and self-adaptive capabilities.

A. Additional Meta-Model Specifications

In the following sections, we give further details about the application architecture and usage profile meta-model specifications that have been mentioned briefly in Section 5.3. These concepts are part of the work of Brosig and the complete specification can be found in Brosig (2014).

A.1. Application Architecture Meta-Model

A.1.1. Service Behavior Descriptions

To describe the performance behavior of a service offered by a software Component, the *application architecture* meta-model introduced by Brosig (2014) and briefly presented in Section 5.3.1 supports multiple (possibly co-existing) behavior abstractions at different levels of granularity (cf. Figure A.1). Figures A.2 to A.4 show the meta-model specifications for the three types of service behavior abstractions.

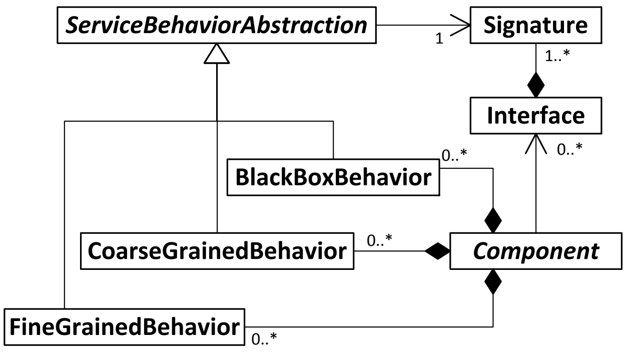


Figure A.1.: Core meta-model specification of the service behavior abstractions.

A BlackBoxBehavior can be described with a ResponseTime characterization. The ResourceDemand captures the total service time required from a given ProcessingResourceType, specified in the resource landscape meta-model. 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 CoarseGrainedBehavior consists of ExternalCallFrequencies and ResourceDemands. An ExternalCallFrequency characterizes the type and the number of external service calls. This

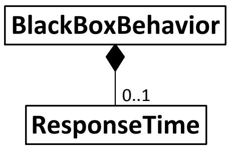


Figure A.2.: Meta-model specifications of the black-box behavior description.

behavior abstraction captures the service behavior when observed from the outside at the service providing component's boundaries. 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.

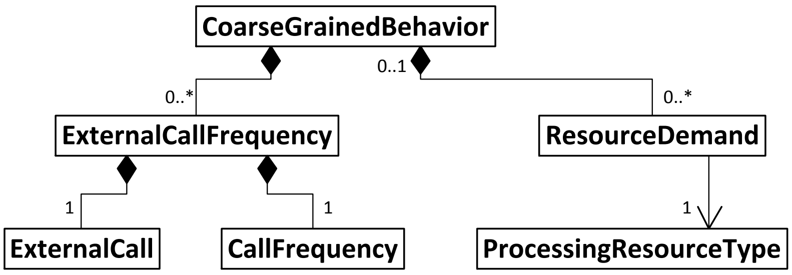


Figure A.3.: Meta-model specifications of the coarse-grained behavior description.

The fine-grained behavior abstraction captures the performance-relevant aspects of the service control flow as abstraction of the actual control flow. Performance-relevant aspects are component-internal computation tasks, the acquisition and release of locks, as well as external service calls, hence also loops and branches where external services are called.

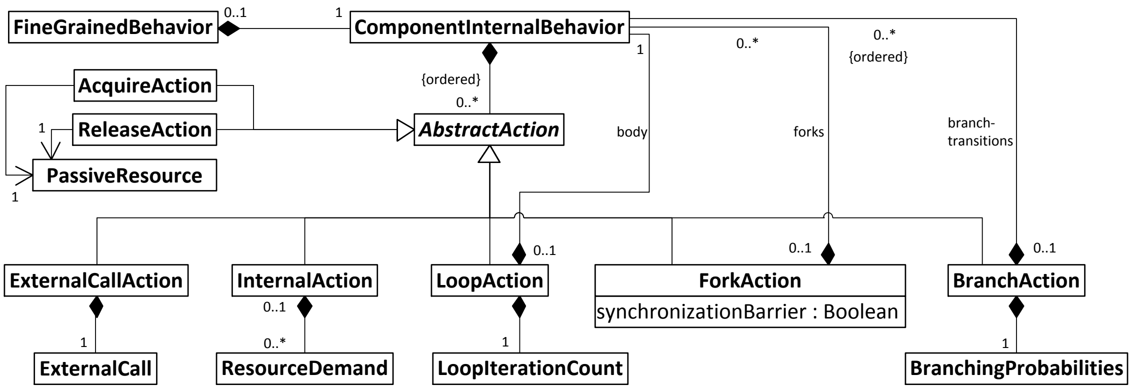


Figure A.4.: Meta-model specifications of the fine-grained behavior description.

The fine-grained behavior abstraction contains a ComponentInternalBehavior that 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. 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. LoopActions and BranchActions can be characterized with loop iteration counts respectively branching probabilities. ForkActions can be modeled either with or without a synchronization barrier. A barrier for the group of the ForkActions forks means that the control flow only proceeds when all forks reached the barrier.

A.1.2. Signature

An Interface of a software Component provides one or more services that are described by their Signature (cf. Figure A.1). A Signature is uniquely defined by its return type and possible input Parameters (cf. Figure A.5). Each Parameter has a name and refers to a DataType.

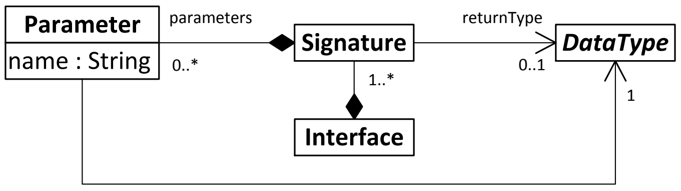


Figure A.5.: Meta-model specification for a signature.

To ensure that the services specified by an interface are unique, we use the OCL constraint given in Listing A.1.

```
context Interface
inv SignaturesMustBeUnique:
  let sigs : Bag(
    Tuple(serviceName : String,
       parameters : Sequence(DataType)))
    = self.signatures
      ->collect(sig : Signature | Tuple{
        serviceName : String = sig.name,
        parameters : Sequence(DataType)
          = sig.parameters.dataType
            ->asSequence()})
  in sigs->isUnique(s|s);
```

Listing A.1: Signatures must be unique.

A.2. Usage Profile Meta-Model

In order to derive performance predictions for certain workloads, the workloads must be specified. The *usage profile* meta-model is intended to allow such specifications of the user interactions with the system. Figure A.6 shows the usage profile modeling abstractions. A UsageProfileModel consists of multiple UsageScenarios. A UsageScenario refers to a description of a WorkloadType (either an open workload or a closed workload) and a ScenarioBehavior. The latter allows to model the control flow of optionally parameterized system calls and delays with branches and loops. Delays, as well as branching probabilities and loop iteration counts, are described explicitly.

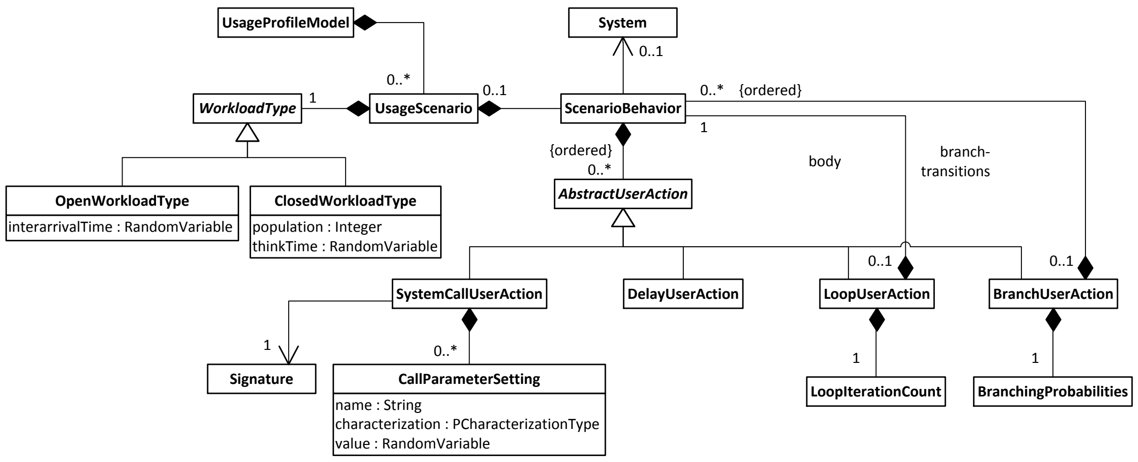


Figure A.6.: Usage profile meta-model.

List of Figures

List of Tables

Bibliography

Abdelzaher, T. F., Shin, K. G., and Bhatti, N. (2002). Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96.

Agrawal, A., Karsai, G., and Shi, F. (2003). A UML-based graph transformation approach for implementing domain-specific model transformations. *Journal on Software and Systems Modeling*, pages 1–19.

Almeida, J., Almeida, V., Ardagna, D., Cunha, I., Francalanci, C., and Trubian, M. (2010). Joint admission control and resource allocation in virtualized servers. *Jour. of Par. and Distr. Comp.*, 70(4):344 – 362.

Almeida, R. and Vieira, M. (2011). Benchmarking the resilience of self-adaptive software systems: perspectives and challenges. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 190–195.

Amazon Web Services (2010). Amazon auto scaling. `http://aws.amazon.com/documentation/autoscaling/`. Last visited: March 2014.

Amoui, M., Derakhshanmanesh, M., Ebert, J., and Tahvildari, L. (2012). Achieving dynamic adaptation via management and interpretation of runtime models. *Journal of Systems and Software*, 85:2720–2737.

Apparao, P., Iyer, R., Zhang, X., Newell, D., and Adelmeyer, T. (2008). Characterization & Analysis of a Server Consolidation Benchmark. In *VEE '08*.

Apparao, P., Makineni, S., and Newell, D. (2006). Characterization of network processing overheads in xen. *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing.*

Assimakopoulos, V. and Nikolopoulos, K. (2000). The theta model: a decomposition approach to forecasting. *International Journal of Forecasting*, 16(4):521 – 530.

Atkinson, C. and Gerbig, R. (2012). Melanie: multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 7:1–7:2, New York, NY, USA. ACM.

Atkinson, C., Gutheil, M., and Kennel, B. (2009). A flexible infrastructure for multilevel language engineering. *IEEE Transactions on Software Engineering*, 35(6):742–755.

Balsamo, S., Di Marco, A., Inverardi, P., and Simeoni, M. (2004). Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. on Software Engineering*, 30(5).

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the 19th Symposium on Operating Systems Principles.*

Becker, M., Luckey, M., and Becker, S. (2012). Model-driven performance engineering of self-adaptive systems: a survey. In *Proceedings of the 8th international ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '12, pages 117–122. ACM.

Becker, M., Luckey, M., and Becker, S. (2013). 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., Koziolek, H., and Reussner, R. (2009). The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22.

Bencomo, N. and Blair, G. (2009). Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 183–200. Springer Berlin Heidelberg.

Bencomo, N., Grace, P., Flores, C., Hughes, D., and Blair, G. (2008). Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. In *Proceedings of the International conference on Software engineering - ICSE '08*, pages 811–814.

Bennani, M. and Menasce, D. (2005). Resource allocation for autonomic data centers using analytic performance models.

Blair, G., Bencomo, N., and France, R. (2009). Models@Run.time. *Computer*, 42(10):22–27.

Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308.

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.

Box, G., Jenkins, G., and Reinsel, G. (2008). *Time series analysis : forecasting and control*. Wiley series in probability and statistics. Wiley, Hoboken, NJ, 4. ed. edition.

Brogan, W. L. (1991). *Modern Control Theory (3rd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Brosig, F. (2014). *Architecture-Level Software Performance Models for Online Performance Prediction*. PhD thesis, Karlsruhe Institute of Technology (KIT). To be published.

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 2011)*.

Brosig, F., Huber, N., and Kounev, S. (2013). 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. ACM, New York, NY, USA.

Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., and Shaw, M. (2009). Engineering self-adaptive systems through feedback loops. In Cheng, B. H., Lemos, R., Giese, H., Inverardi, P., and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer-Verlag, Berlin, Heidelberg.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK.

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 ACM SIGMETRICS International Conference.*

Cheng, B. H. C., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. (2009). Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Cheng, B. H. C., Lemos, R., Giese, H., Inverardi, P., and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg.

Cheng, S.-W. and Garlan, D. (2012). Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860 – 2875.

Cheng, S.-W., Garlan, D., and Schmerl, B. (2006). Architecture-based self-adaptation in the presence of multiple objectives. In *International Workshop Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 2–8. ACM.

Cunha, I., Almeida, J., Almeida, V., and Santos, M. (2007). Self-adaptive capacity management for multi-tier virtualized environments. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, IM '07, pages 129–138.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming.* Addison-Wesley.

da Silva, C. and de Lemos, R. (2009). Using dynamic workflows for coordinating self-adaptation of software systems. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 86–95. IEEE.

de Lemos, R., Giese, H., Müller, H., and Shaw, M. (2011). Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, number 10431 in Dagstuhl Seminar Proceedings.

De Livera, A. M., Hyndman, R. J., and Snyder, R. D. (2011). Forecasting time series with complex seasonal patterns using exponential smoothing. *Journal of the American Statistical Association*, 106(496):1513–1527.

Descartes, R. (1644). *Principia philosophiae.* apud Ludovicum Elzevirium.

Descartes Research Group (2013). Tools of the Descartes Research Group. `http://www.descartes-research.net/tools/`.

Esfahani, N., Malek, S., Sousa, J., Gomaa, H., and Menascé, D. (2009). A modeling language for activity-oriented composition of service-oriented software systems. In *Int. Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 591–605. ACM.

Eskenazi, E., Fioukov, A., and Hammer, D. (2004). Performance Prediction for Component Compositions. In *Proceedings of the International Symposium on Component-Based Software Engineering.*

Eucalyptus Systems Inc. (2013). Eucalyptus — Open Source AWS Compatible Private Clouds. `http://www.eucalyptus.com/`.

Fischer, T., Niere, J., Torunski, L., and Zündorf, A. (2000). Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. *Theory and Application of Graph Transformations*, pages 296–309.

Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjorven, E. (2006). Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70.

France, R. and Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07).*

Franklin, S. and Graesser, A. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In Müller, J. P., Wooldridge, M. J., and Jennings, N. R., editors, *Intelligent Agents III Agent Theories, Architectures, and Languages*, volume 1193 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin Heidelberg.

Franks, G., Majumdar, S., Neilson, Petriu, D., Rolia, J., and Woodside, M. (1996). Performance analysis of distributed server systems. In *Proceedings of the 6th International Conference on Software Quality*, pages 15–26.

Franks, G., Maly, P., Woodside, M., Petriu, D. C., and Hubbard, A. (2011). Layered queueing network solver and simulator user manual. `http://www.sce.carleton.ca/rads/lqns/LQNSUserMan.pdf`. Real-time and Distributed Systems Lab, Carleton University, Ottawa.

Frey, S., van Hoorn, A., Jung, R., Kiel, B., and Hasselbring, W. (2012). MAMBA: Model-Based Software Analysis Utilizing OMG's SMM. In *Proceedings of the 14. Workshop Software-Reengineering (WSR '12)*, pages 37–38. Also appeared in Softwaretechnik-Trends 32(2) (May 2012) 49-50.

Gambi, A., Toffetti, G., Pautasso, C., and Pezze, M. (2013). Kriging Controllers for Cloud Applications. *IEEE Internet Computing*, 17(4):40–47.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. (2004). Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54.

Garlan, D., Schmerl, B., and Cheng, S.-W. (2009). Software architecture-based self-adaptation. In *Autonomic Computing and Networking*, pages 31–55. Springer US.

Geihs, K., Barone, P., Eliassen, F., Floch, J., Fricke, R., Gjorven, E., Hallsteinsen, S., Horn, G., Khan, M. U., Mamelli, A., Papadopoulos, G. A., Paspallis, N., Reichle, R., and Stav, E. (2009). A comprehensive solution for application-level adaptation. *Software Practice & Experience*, 39:385–422.

Gilmore, S., Haenel, V., Kloul, L., and Maidl, M. (2005). Choreographing Security and Performance Analysis for Web Services. In *Formal Techniques for Computer Systems and Business Processes.*

Goodwin, P. (2010). The Holt-Winters Approach to Exponential Smoothing: 50 Years Old and Going Strong. *FORESIGHT: The International Journal of Applied Forecasting, Forthcoming.* Available at SSRN: `http://ssrn.com/abstract=1714811`.

Gorlick, M. M. and Razouk, R. R. (1991). Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE '91, pages 23–34, Los Alamitos, CA, USA. IEEE Computer Society Press.

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. (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.

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.

Hallsteinsen, S., Geihs, K., Paspallis, N., Eliassen, F., Horn, G., Lorenzo, J., Mamelli, A., and Papadopoulos, G. (2012). A development framework and methodology for self-adapting applications in ubiquitous computing environments. *Journal of Systems and Software*, 85:2840–2859.

Herbst, N. R. (2012). Workload Classification and Forecasting. Diploma Thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany.

Herbst, N. R., Huber, N., Kounev, S., and Amrehn, E. (2013a). Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 187–198, 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.*

Herbst, N. R., Kounev, S., and Reussner, R. (2013b). Elasticity in Cloud Computing: What it is, and What it is Not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013)*. USENIX.

Horn, P. (2001). Autonomic Computing: IBM's Perspective on the State of Information Technology.

Huber, N., Becker, S., Rathfelder, C., Schweflinghaus, J., and Reussner, R. (2010a). Performance Modeling in Industry: A Case Study on Storage Virtualization. In *ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010), Software Engineering in Practice Track*, pages 1–10, New York, NY, USA. ACM.

Huber, N., Brosig, F., and Kounev, S. (2011a). Model-based Self-Adaptive Resource Allocation in Virtualized Environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'11, 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 International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA'12, 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 *IEEE 9th International Conference on e-Business Engineering*, ICEBE'12. IEEE.

Huber, N., van Hoorn, A., Koziolek, A., Brosig, F., and Kounev, S. (2014). Modeling Run-Time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments. *Service Oriented Computing and Applications Journal (SOCA)*, 8(1):73–89.

Huber, N., Von Quast, M., Brosig, F., and Kounev, S. (2010b). Analysis of the Performance-Influencing Factors of Virtualization Platforms. In *Proceedings of the 2010 International Conference on On the move to meaningful internet systems: Part II*, OTM'10, pages 811–828, Berlin, Heidelberg. Springer-Verlag.

Huber, N., von Quast, M., Hauck, M., and Kounev, S. (2011b). Evaluating and modeling virtualization performance overhead for cloud environments. In Leymann, F., Ivanov, I., van Sinderen, M., and Shishkov, B., editors, *Proceedings of the International Conference on Cloud Computing and Services Science*, CLOSER'11, pages 563–573. SciTePress.

Huebscher, M. C. and McCann, J. A. (2008). A Survey of Autonomic Computing — Degrees, Models, and Applications. *ACM Computing Surveys*, 40(3):1–28.

Hyndman, R. (2008). *Forecasting with Exponential Smoothing : The State Space Approach.* Springer Series in Statistics. Springer-Verlag Berlin Heidelberg.

Hyndman, R. J. (2009). Forecast package for R. `http://robjhyndman.com/software/forecast/`. v3.2.

Hyndman, R. J. and Khandakar, Y. (2008). Automatic time series forecasting: The forecast package for R.

Hyndman, R. J., King, M. L., Pitrun, I., and Billah, B. (2002). Local linear forecasts using cubic smoothing splines. Monash Econometrics and Business Statistics Working Papers 10/02, Monash University, Department of Econometrics and Business Statistics.

Hyndman, R. J. and Koehler, A. B. (2006). Another look at measures of forecast accuracy. *International Journal of Forecasting*, pages 679–688.

IBM Corporation (2003). An architectural blueprint for autonomic computing (IBM White Paper). Technical report.

Iometer (2006). `http://www.iometer.org/`. Version 2006.07.27, last accessed 2011.

Iperf (2003). `http://sourceforge.net/projects/iperf/`. v1.7.0, last accessed 2011.

IT world, The IDG Network (2008). Gartner's data on energy consumption, virtualization, cloud. http://www.itworld.com/green-it/59328/gartners-data-energy-consumption-virtualization-cloud.

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.

Jung, G., Hiltunen, M. A., Joshi, K. R., Schlichting, R. D., and Pu, C. (2010). Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 62–73, Washington, DC, USA. IEEE Computer Society.

Jung, G., Joshi, K., Hiltunen, M. A., Schlichting, R. D., and Pu, C. (2008). Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments. In *Proceedings of the 2008 International Conference on Autonomic Computing.*

Kaplan, J., Forrest, W., and Kindler, N. (2008). Revolutionizing Data Center Energy Efficiency. McKinsey.

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 *Proceedings of the 4th International Conference on Autonomic Computing* .

Kephart, J. and Walsh, W. (2004). An artificial intelligence perspective on autonomic computing policies. In *International Workshop on Policies for Distributed Systems and Networks*, pages 3–12. IEEE.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kephart, J. O., Kounev, S., Kwiatkowska, M., and Zhu, X. (2015). Model-driven Algorithms and Architectures for Self-Aware Computing Systems. Dagstuhl Seminar 15041, `http://www.dagstuhl.de/15041`.

Kounev, S. (2006). Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Trans. on Softw. Eng.*, 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 2011).*

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.

Koziolek, A. and Reussner, R. (2011). Towards a generic quality optimisation framework for component-based system models. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component-Based Software Engineering*, CBSE '11, pages 103–108, New York, NY, USA. ACM.

Koziolek, H. (2010). Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658.

Kramer, J. and Magee, J. (2007). Self-managed systems: an architectural challenge. In *Future of Software Engineering 2007, FOSE '07*, pages 259–268.

Lalanda, P., McCann, J. A., and Diaconescu, A. (2013). *Autonomic Computing.* Springer.

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.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, UK. Springer-Verlag.

Martens, A., Koziolek, H., Becker, S., and Reussner, R. H. (2010). Automatically improve software models for performance, reliability and cost using genetic algorithms. In *Int. Conference on Performance Engineering (ICPE)*, pages 105–116. ACM.

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 *IEEE International Conference on Services Computing.*

Microsoft (2012). Autoscaling and Windows Azure. `http://msdn.microsoft.com/en-us/library/hh680945(v=pandp.50).aspx`. Last built: June 7, 2012.

Morin, B., Barais, O., Jezequel, J., Fleurey, F., and Solberg, A. (2009). Models@run.time to support dynamic adaptation. *Computer*, 42(10):44–51.

Noorshams, Q., Bruhn, D., Kounev, S., and Reussner, R. (2013a). Predictive Performance Modeling of Virtualized Storage Systems using Optimized Statistical Regression Techniques. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE'13, pages 283–294, New York, NY, USA. ACM.

Noorshams, Q., Rentschler, A., Kounev, S., and Reussner, R. (2013b). A Generic Approach for Architecture-level Performance Modeling and Prediction of Virtualized Storage Systems. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE'13, pages 339–342, New York, NY, USA. ACM.

Noorshams, Q., Rostami, K., Kounev, S., Tůma, P., and Reussner, R. (2013c). I/O Performance Modeling of Virtualized Storage Systems. In *Proceedings of the IEEE 21st International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS'13. IEEE.

Object Management Group (OMG) (2005). UML Profile for Schedulability, Performance, and Time (SPTP). `http://www.omg.org/spec/SPTP/1.1/`.

Object Management Group (OMG) (2011a). Business Process Model And Notation (BPMN). `http://www.omg.org/spec/BPMN/2.0/`.

Object Management Group (OMG) (2011b). Meta Object Facility (MOF) Core. `http://www.omg.org/spec/MOF/2.4.1/`.

Object Management Group (OMG) (2011c). UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). `http://www.omg.org/spec/MARTE/1.1/`.

Object Management Group (OMG) (2012). Structured Metrics Meta-Model (SMM). `http://www.omg.org/spec/SMM/1.0/`.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62.

Padala, P., Zhu, X., Wang, Z., Singhal, S., and Shin, K. G. (2007). Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report.*

Passmark PerformanceTest (2009). `http://www.passmark.com/products/pt.htm`. v7.0, last accessed 2011.

Patikirikorala, T., Colman, A., Han, J., and Wang, L. (2012). An evaluation of multi-model self-managing control schemes for adaptive performance management of software systems. *Journal of Systems and Software*, 85(12):2678–2696.

PCM Bench (2014). `http://www.palladio-simulator.org`. v3.4.1, last accessed 2014.

Petriu, D. and Woodside, M. (2007). An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Softw. and Syst. Modeling*, 6(2):163–184.

Quétier, B., Néri, V., and Cappello, F. (2007). Scalability Comparison of Four Host Virtualization Tools. *Jounal on Grid Computing*, 5(1):83–98.

R Project (2013). R, a language and environment for statistical computing and graphics. `http://www.r-project.org/`. v2.15.0.

Rattu, D. (2012). Infrastructure Reconfiguration Using Architecture-Level Performance Models. Master's thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany.

Rosenblum, M. and Garfinkel, T. (2005). Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47.

Rygielski, P., Kounev, S., and Zschaler, S. (2013a). Model-Based Throughput Prediction in Data Center Networks. In *Proceedings of the 2nd IEEE International Workshop on Measurements and Networking (M&N 2013)*. IEEE.

Rygielski, P., Zschaler, S., and Kounev, S. (2013b). A Meta-Model for Performance Modeling of Dynamic Virtualized Network Infrastructures. In *International Conference on Performance Engineering (ICPE)*.

Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42.

Schott, W. (2013). Automated Model-based System Reconfiguration: A Case Study. Master's thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany.

Shenstone, L. and Hyndman, R. J. (2005). Stochastic models underlying croston's method for intermittent demand forecasting. *Journal of Forecasting*, 24(6):389–402.

Shumway, R. H. (2011). *Time Series Analysis and Its Applications : With R Examples.* Springer Science+Business Media, LLC, New York, NY.

Smith, C. and Williams, L. G. (2002). *Performance solutions: a practical guide to creating responsive, scalable software.* Addison Wesley.

Smith, C. U. (1981). Increasing Information Systems Productivity by Software Performance Engineering. In *International CMG Conference*, pages 5–14.

Smith, C. U., Lladó, C. M., Cortellessa, V., Marco, A. D., and Williams, L. G. (2005). From uml models to software performance results: An spe process based on xml interchange formats. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, pages 87–98, New York, NY, USA. ACM.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287.

SPEC CPU2006 (2006). `http://www.spec.org/cpu2006/`. v1.1, last accessed 2011.

Spinner, S., Kounev, S., and Meier, P. (2012). Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0. In *Int. Conference on Application and Theory of Petri Nets and Concurrency*, volume 7347, pages 388–397.

Sykes, D., Heaven, W., Magee, J., and Kramer, J. (2008). From goals to components: A combined approach to self-management. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 1–8, New York, NY, USA. ACM.

Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 2nd edition.

Tajalli, H., Garcia, J., Edwards, G., and Medvidovic, N. (2010). Plasma: A plan-based layered architecture for software model-driven adaptation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 467–476, New York, NY, USA. ACM.

Taylor, R., Medvidovic, N., Anderson, K. M., Whitehead Jr., E., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. (1996). A component- and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390–406.

Tesauro, G., Jong, N. K., Das, R., and Bennani, M. N. (2006). A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *IEEE International Conference on Autonomic Computing*.

Tickoo, O., Iyer, R., Illikkal, R., and Newell, D. (2010). Modeling virtual machine performance: Challenges and approaches. volume 37, pages 55–60, New York, NY, USA. ACM.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM SIGMETRICS International Conference*.

van der Aalst, W. and ter Hofstede, A. (2005). YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275.

van Hoorn, A. (2014). *Online Capacity Management for Increased Resource Efficiency of Component-Based Software Systems*. PhD thesis, University of Kiel, Germany. Work in progress.

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.

van Hoorn, A., Waller, J., and Hasselbring, W. (2012). Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM.

Verbesselt, J. (2009). Breaks for additive season and trend project. `http://bfast.r-forge.r-project.org/`.

Verbesselt, J., Hyndman, R., Zeileis, A., and Culvenor, D. (2010). Phenological change detection while accounting for abrupt and gradual trends in satellite image time series. *Remote Sensing of Environment*, 114(12):2970 – 2980.

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*.

Villegas, N. M., Tamura, G., Müller, H. A., Duchien, L., and Casallas, R. (2013). DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*, pages 265–293.

VMware (2006). Resource Management with VMware DRS. `http://www.vmware.com/pdf/vmware_drs_wp.pdf`. Latest revision: June 5, 2006.

VMware (2007). A performance comparison of hypervisors. `http://www.vmware.com/pdf/hypervisor_performance.pdf`. Latest revision: March, 2014.

Vogel, T. and Giese, H. (2012). Requirements and assessment of languages and frameworks for adaptation models. In *International Conference on Models in Software Engineering (MODELS)*, pages 167–182. Springer.

Vogel, T. and Giese, H. (2014). Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions Autonomous and Adaptive Systems*, 8(4):18:1–18:33.

Vogel, T., Seibel, A., and Giese, H. (2011). The Role of Models and Megamodels at Runtime. In *Models in Software Engineering*, pages 224–238.

von Massow, R., van Hoorn, A., and Hasselbring, W. (2011). Performance simulation of runtime reconfigurable component-based software architectures. In *European Conference on Software Architecture (ECSA)*, pages 43–58. Springer.

Weyns, D., Malek, S., and Andersson, J. (2012). FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions Autonomous and Adaptive Systems*, 7(1):8.

Widom, J. and Ceri, S., editors (1996). *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann.

Wikimedia Project (2011). Page view statistics for Wikimedia projects. `http://dumps.wikimedia.org/other/pagecounts-raw/`. Accessed: October 2011.

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 4th International Conference on Autonomic Computing*, ICAC '07, Washington, DC, USA. IEEE Computer Society.