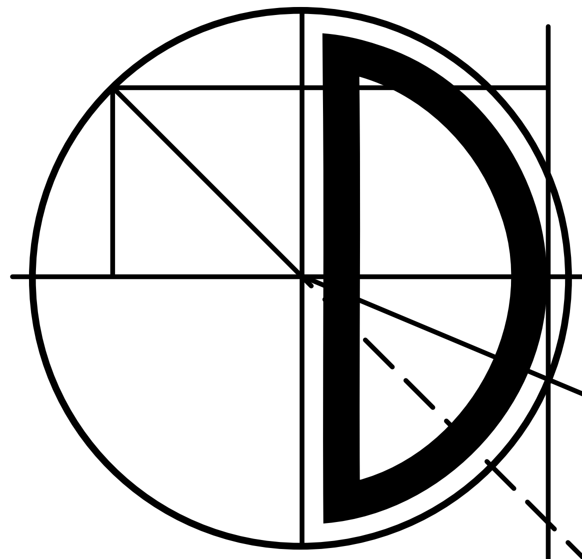
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Descartes research

The Descartes Modeling Language



Samuel Kounev, Fabian Brosig, Nikolaus Huber

Descartes Research Group
Chair of Software Engineering
Department of Computer Science
University of Würzburg, Germany

`{samuel.kounev|fabian.brosig|nikolaus.huber}@uni-wuerzburg.de`

October 13, 2014
`v1.0`

CC BY SA

Contents

1 Introduction 6
- 1.1 Motivation . . . 6
- 1.2 Design-time vs. Run-Time Models . . . 8
- 1.3 The Descartes Modeling Language (DML) . . . 11
  - 1.3.1 Modeling Language Overview . . . 11
  - 1.3.2 Summary of Supported Features and Novel Aspects . . . 13
  - 1.3.3 Application Scenarios . . . 14
- 1.4 Self-Aware Computing Systems . . . 17
- 1.5 Outline . . . 18

2 Background 19
- 2.1 Performance Modeling Approaches . . . 19
  - 2.1.1 Existing Architecture-Level Performance Models . . . 19
  - 2.1.2 Palladio Component Model (PCM) . . . 20
- 2.2 Modeling Run-time System Adaptation . . . 23
  - 2.2.1 Abstraction Levels . . . 23
  - 2.2.2 Languages for Adaptation Control Flow . . . 23
  - 2.2.3 Configuration Space . . . 24

3 Online Performance Prediction Scenario 25
- 3.1 Setting . . . 25
- 3.2 SPECjEnterprise2010 . . . 26
- 3.3 Exemplary Environment . . . 27

4 Architecture-Level Performance Model 29
- 4.1 Application Architecture Model . . . 29
  - 4.1.1 Component Model and System Model . . . 29
  - 4.1.2 Running Example . . . 32
  - 4.1.3 Service Behavior Abstractions . . . 32
  - 4.1.4 Parameterization . . . 38
  - 4.1.5 Probabilistic Parameter Dependencies . . . 42
  - 4.1.6 Interface to Monitoring Infrastructure . . . 49
- 4.2 Resource Landscape Model . . . 51
  - 4.2.1 Modeling Abstractions . . . 53
  - 4.2.2 Example . . . 59
- 4.3 Deployment Model . . . 60
  - 4.3.1 Modeling Abstractions . . . 60
  - 4.3.2 Example . . . 61

4.4 Usage Profile Model . . . 62
4.4.1 Modeling Abstractions . . . 62
4.4.2 Example . . . 63

5 Model-based System Adaptation 64
5.1 Motivation and Background . . . 64
5.2 Adaptation Points Model . . . 66
5.3 Adaptation Process Model . . . 70
5.3.1 Actions . . . 72
5.3.2 Tactics . . . 73
5.3.3 Strategies . . . 76
5.3.4 QoS Data Repository . . . 77
5.3.5 Weighting Function . . . 78

6 Discussion 81
6.1 Differences between DML and PCM . . . 81
6.2 Ongoing and Future Work . . . 82

Bibliography 85

List of Acronyms and Abbreviations 93

List of Figures

1.1 Degrees-of-Freedom and performance-influencing factors in a modern IT system. 7
1.2 Relation of the different models of a Descartes Modeling Language (DML) instance . . . 11
1.3 Model-Based System Adaptation Control Loop [22] 16

2.1 Components providing and requiring interfaces. 21
2.2 Assembly of a composite component. 21
2.3 RD-SEFF of service with signature *execute(int number, List array)* (cf. [11]). 22

3.1 Online performance prediction scenario. 25
3.2 SPECjEnterprise2010 architecture [48]. 26
3.3 Experimental environment. 27

4.1 Components and Interfaces, cf. [11] 30
4.2 Component Type Hierarchy, cf. [11] 30
4.3 Component Composition, cf. [11] 31
4.4 Example: System Instance as UML Object Diagram 31
4.5 Example: System Instance 32
4.6 (a) Composition Tree Schema and (b) Example System Instance as Composition Tree . . 33
4.7 Component Instance Reference 33
4.8 Running Example: WebShop 33
4.9 Example: *Delivery* Component 34
4.10 Different Service Behavior Abstractions 36
4.11 (a) Coarse-Grained and (b) Black-Box Behavior Abstractions 36
4.12 Fine-Grained Behavior Abstraction, cf. [11] 37
4.13 Example: *Delivery* and *ShoppingCartServlet* 37
4.14 Example: Fine-Grained Behavior Abstraction of Service *calculateTotalCost* Provided by *ShoppingCartServlet* 38
4.15 Example: Coarse-Grained Behavior Abstraction of Service *calculateTotalCost* Provided by *ShoppingCartServlet* 38
4.16 Example: WebShops for a *GameStore* and a *Supermarket* 39
4.17 Model Variables 40
4.18 Example: *CatalogServlet* and *JPAProvider* Components 42
4.19 Example: Cache Miss or Cache Hit in Service *getArticlePreviewImage* 43
4.20 Example: Behavior of *listArticles* Service Provided by *CatalogServlet* 43
4.21 Modeling Parameter Dependencies 44
4.22 Influenced Variables and Influencing Parameters 46
4.23 Call Parameter Hierarchy 46
4.24 Call Parameters 47

4.25 Relationships between Influenced Variables and Influencing Parameters 47
4.26 Characterization of Relationships 48
4.27 Example: Modeling Parameter Dependencies 49
4.28 Example: Characterizing Parameter Dependencies 50
4.29 Main types of data center resources. 52
4.30 Different resource layers and their influence on the performance. 53
4.31 The resource landscape meta-model. 54
4.32 Different runtime environment classes. 55
4.33 Types of resource configurations. 56
4.34 Container instances in the MOF modeling hierarchy 58
4.35 The container templates repository. 58
4.36 Example resource landscape model instance. 59
4.37 The deployment meta-model. 60
4.38 Example: WebShop Deployment 61
4.39 Usage Profile Model, cf. [11] 62
4.40 Example: Usage Profile Model Instance 63

5.1 Interaction of the system, the system models and the S/T/A adaptation language. 65
5.2 Relation of Entity and AdaptableEntity. 66
5.3 Adaptation points meta-model. 68
5.4 Adaptation points meta-model instance. 69
5.5 Concepts of the adaptation process meta-model and their relations. 71
5.6 Adaptation process meta-model. 72
5.7 Example Actions referring to adaptation points. 73
5.8 Different example Tactics using the previously specified Actions. 74
5.9 Example Strategies using Tactics with assigned weights. 77
5.10 The QoS data repository meta-model. 78

Chapter 1

Introduction

This technical report introduces the Descartes Modeling Language (DML), a new architecture-level modeling language for modeling Quality-of-Service (QoS) and resource management related aspects of modern dynamic IT systems, infrastructures and services. DML is designed to serve as a basis for *self-aware* resource management[1] [1, 2] during operation ensuring that system quality-of-service requirements are continuously satisfied while infrastructure resources are utilized as efficiently as possible. The term Quality-of-Service (QoS) is used to refer to non-functional system properties including performance (considering classical metrics such as response time, throughput, scalability and efficiency) and dependability (considering in addition: availability, reliability and security aspects). The current version of DML is focused on performance and availability including capacity, responsiveness and resource efficiency aspects, however, work is underway to provide support for modeling further QoS properties. The meta-model itself is designed in a generic fashion and is intended to eventually support the full spectrum of QoS properties mentioned above. Given that the initial version of DML is focussed on performance, in the rest of this document, we mostly speak of performance instead of QoS in general. Information on the latest developments around the Descartes Modeling Language (DML) can be found at `http://www.descartes-research.net`.

1.1 Motivation

Modern IT systems have increasingly complex and dynamic architectures composed of loosely-coupled distributed components and services that operate and evolve independently. Managing system resources in such environments to ensure acceptable end-to-end application QoS while at the same time optimizing resource utilization and energy efficiency is a challenge [3, 4, 5]. The adoption of virtualization and cloud computing technologies, such as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS), comes at the cost of increased system complexity and dynamicity.

The increased complexity is caused by the introduction of virtual resources and the resulting gap between logical and physical resource allocations. The increased dynamicity is caused by the complex interactions between the applications and workloads sharing the physical infrastructure. The inability to predict such interactions and adapt the system accordingly makes it hard to provide QoS guarantees in terms of availability and responsiveness, as well as resilience to attacks and operational failures [6]. Moreover, the consolidation of workloads translates into higher utilization of physical resources which makes systems much more vulnerable to threats resulting from unforeseen load fluctuations, hardware failures and network attacks.

System administrators and service providers are often faced with questions such as:

[1]The interpretation of the term "self-aware" is described in detail in Sec. 1.4

- What QoS would a new service or application deployed on the virtualized infrastructure exhibit and how much resources should be allocated to it?
- How should the workloads of the new service/application and existing services be partitioned among the available resources so that QoS requirements are satisfied and resources are utilized efficiently?
- What would be the effect of adding a new component or upgrading an existing component as services and applications evolve?
- If an application experiences a load spike or a change of its workload profile, how would this affect the system QoS? Which parts of the system architecture would require additional resources?
- At what granularity and at what rate should resources be provisioned / released as workloads fluctuate (e.g., CPU time, virtual cores, virtual machines, physical servers, clusters, data centers)?
- What would be the effect of migrating a service or an application component from one physical server to another?
- How should the system configuration (e.g., component deployment, resource allocations) be adapted to avoid inefficient system operation arising from evolving application workloads?

Answering such questions requires the ability to predict at *run-time* how the QoS of running applications and services would be affected if application workloads change and/or the system deployment and configuration is changed. We refer to this as *online QoS prediction*. Given that the initial version of DML is focussed on performance, hereafter we will speak of *online performance prediction* [1, 2].

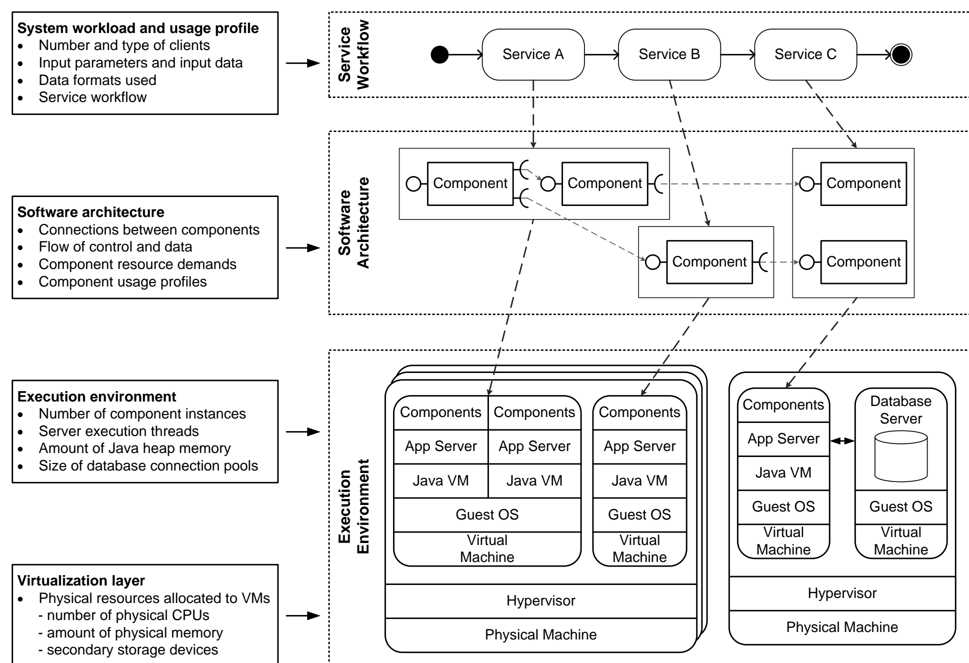


Figure 1.1: Degrees-of-Freedom and performance-influencing factors in a modern IT system.

Predicting the performance of a modern IT system, however, even in an offline scenario is a challenging task. Consider the architecture of a typical modern IT system as depicted in Figure 1.1. For a given set of hardware and software platforms at each layer of the architecture, Figure 1.1 shows some examples of the degrees-of-freedom at each layer and the factors that may affect the performance of the system. Predicting the performance of a service requires taking these factors into account as well as the

dependencies among them. For example, the input parameters passed to a service may have direct impact on the set of software components involved in executing the service, as well as their internal behavior (e.g., flow of control, number of loop iterations, return parameters) and resource demands (e.g., CPU, disk and network service times). Consider for instance an online translation service. The time needed to process a translation request and the specific system components involved would depend on the size of the document passed as input, the format in which the document is provided, as well as the source and target languages. Thus, in order to predict the service response time, the effects of input parameters have to be traced through the complex chain of components and resources involved. Moreover, the configuration parameters at the different layers of the execution environment, as well as resource contention due to concurrently executed requests, must be taken into account. Therefore, a detailed *performance model* capturing the performance-relevant aspects of both the software architecture and the multi-layered execution environment is needed.

Existing approaches to online performance prediction (e.g., [7, 8, 9, 10]) are based on stochastic performance models such as queueing networks, stochastic petri nets and variants thereof, e.g., layered queueing networks or queueing petri nets. Such models, often referred to as *predictive* performance models, normally abstract the system at a high level without explicitly taking into account its software architecture (e.g., flow of control and dependencies between software components), its execution environment and configuration (e.g., resource allocations at the virtualization layer). Services are typically modeled as black boxes and many restrictive assumptions are often imposed such as a single workload class, single-threaded components, homogeneous servers or exponential request inter-arrival times. Detailed models that explicitly capture the software architecture, execution environment and configuration exist in the literature, however, such models are intended for offline use at system design time (e.g., [11, 12, 13, 14]). Models in this area are *descriptive* in nature, e.g., software architecture models based on UML, annotated with descriptions of the system's performance-relevant behavior. Such models, often referred to as *architecture-level* performance models, are built during system development and are used at design and/or deployment time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes.

While architecture-level performance models provide a powerful tool for performance prediction, they are typically expensive to build and provide limited support for reusability and customization which renders them impractical for use at run-time. Recent efforts in the area of *component-based performance engineering* [15] have contributed a lot to facilitate model reusability, however, there is still much work to be done on further parameterizing performance models before they can be used for online performance prediction.

1.2 Design-time vs. Run-Time Models

We argue that there are some fundamental differences between offline and online scenarios for performance prediction leading 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. In the following, we summarize the main differences in terms of goals and underlying assumptions driving the evolution of design-time vs. run-time models.

Goal: Evaluate Design Alternatives vs. Evaluate Impact of Dynamic Changes At system design-time, the main goal of performance modeling and prediction is to evaluate and compare different design alternatives in terms of their performance properties.

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

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

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

Type and Amount of Data Available for Model Parameterization and Calibration Performance models typically have multiple parameters such as workload profile parameters (workload mix and workload intensity), resource demands, branch probabilities and loop iteration frequencies. The type and amount of data available as a basis for model parameterization and calibration at design-time vs. run-time greatly differs.

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

At run-time, all system components are implemented and deployed in the target production environment. This makes it possible to obtain much more accurate estimates of the various model parameters taking into account the real execution environment. Moreover, model parameters can be continuously calibrated to iteratively refine their accuracy. Furthermore, performance-relevant information can be

monitored and described at the component instance level, not only at the type level as typical for design-time models. However, during operation, we don't have the possibility to run arbitrary experiments since the system is in production and is used by real customers placing requests. In such a setting, monitoring has to be handled with care, keeping the monitoring overhead within limits (non-intrusive approach) such that system operation is not disturbed. Thus, at run-time, while theoretically much more accurate estimates of model parameters can be obtained, one has less control over the system to run experiments and monitoring must be performed with care in a non-intrusive manner.

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

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

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

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

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

Design for Use by Humans vs. Machines Design-time models are normally designed to be used by humans. They also serve as architecture documentation, i.e., they should be easy to understand and model instances should be valid and meaningful.

In contrast, run-time models are typically used for optimizing the system configuration and deployment as part of autonomic run-time resource management techniques. In this case, models are used by programs or agents as opposed to humans. Ideally, models should be composed automatically at run-time and tailored to the specific prediction scenario taking into account timing constraints and requirements concerning accuracy. Also, ideally, models will be hidden behind the scenes and no users or administrators will ever have to deal with them. Although, in many cases the initial sub-meta-models capturing the performance-relevant aspects of the various system layers would have to be constructed manually,

novel automated model inference techniques increasingly enable the extraction of sub-meta-models in an automatic or semi-automatic manner.

1.3 The Descartes Modeling Language (DML)

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 [16], DML can support system analysis and problem detection as well as 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. In the following section, we give an overview of the different sub-models of DML before its features are explained in detail in Chapter 4 to Chapter 5.

1.3.1 Modeling Language Overview

The Descartes Modeling Language (DML) is a novel architecture-level modeling language to describe Quality of Service (QoS) and resource management related aspects of modern dynamic IT systems, infrastructures and services. 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 1.2). 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).

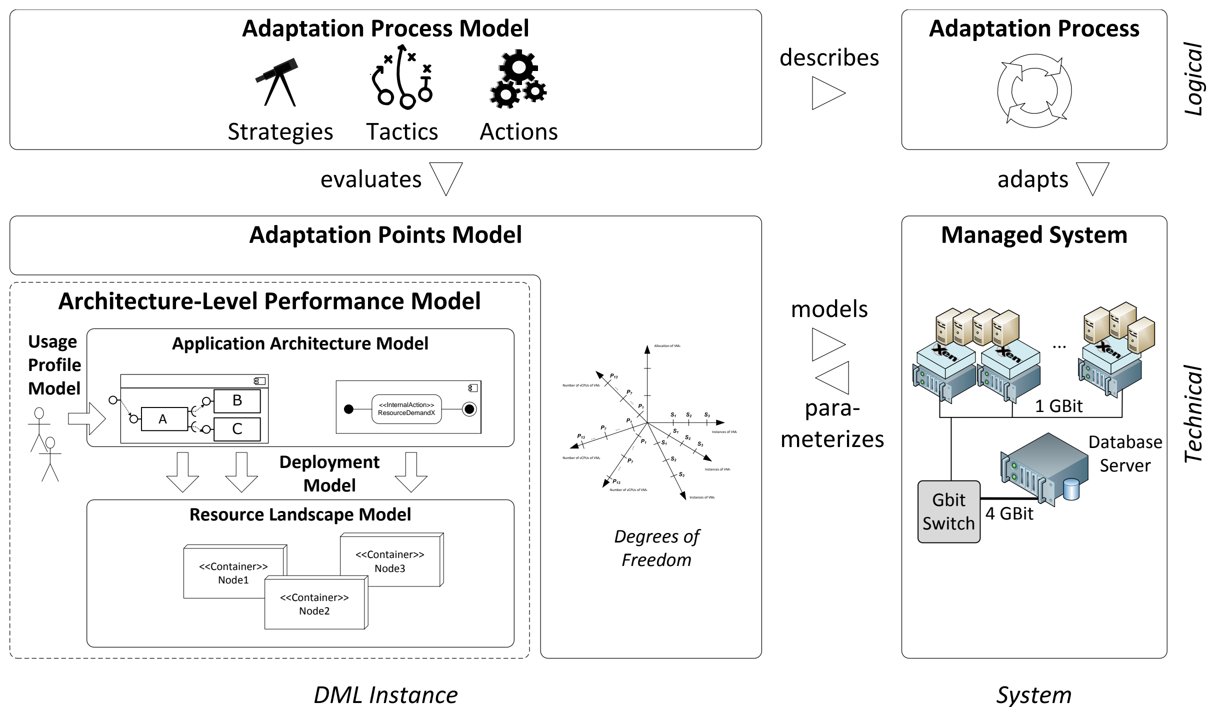


Figure 1.2: Relation of the different models of a DML instance and the system.

Figure 1.2 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. In the bottom right corner of Figure 1.2, we see the system that is managed by a given, usually system-specific, adaptation process, depicted in the top right corner of Figure 1.2. 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). On top of these models (top left corner of Figure 1.2), 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.

In the following paragraphs we give brief overviews of the features of each meta-model.

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 [11]. A software component is defined as a unit of composition with explicitly defined provided and required interfaces [17]. 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.

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.

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.

Usage Profile Model 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.

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 Central Processing Units (vCPUs) to VMs, migrating VMs or software components, or load-balancing requests). Thus, the adaptation points model defines the configuration space of the adapted 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.

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.

1.3.2 Summary of Supported Features and Novel Aspects

The Descartes Modeling Language (DML) provides a new architecture-level modeling language for modeling quality-of-service and resource management related aspects of modern dynamic IT systems, infrastructures and services. DML models can be used both in offline and online settings spanning the whole lifecycle of an IT system. In an offline setting the increased flexibility provided by DML can be exploited for system sizing and capacity planning as well as for evaluating alternative system architectures or target deployment platforms. It can also be used to predict the effect of changes in the

system architecture, deployment and configuration as services and applications evolve. In an online setting, DML provides the basis for *self-aware* resource management during operation ensuring that system quality-of-service requirements are continuously satisfied while infrastructure resources are utilized as efficiently as possible.

From the scientific perspective, the key features of DML are: i) a domain-specific language designed for modeling the performance-relevant behavior of services in dynamic environments, ii) a modeling approach to characterize parameter and context dependencies based on online monitoring statistics, iii) a domain-specific language to model the distributed and dynamic resource landscape of modern data centers capturing the properties relevant for performance and resource management, iv) an adaptation points meta-model for annotating system architecture QoS models to describe the valid configuration space of the modeled dynamic system. v) a modeling language to describe system adaptation strategies and heuristics independent of the system-specific details.

1.3.3 Application Scenarios

The developed performance modeling and prediction approach has been designed to be applicable in different scenarios. As mentioned above, while the major application of DML is to serve as a basis for engineering self-aware software systems (Sec. 1.4), here in this subsection, we provide an overview of more fine-grained application areas.

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

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

Impact Analysis of Workload Changes In general, the workload intensity of enterprise software systems varies over time. The workload intensity may follow certain trends or patterns, e.g., a weekly pattern with low intensity over the weekend. In addition, there can be situations where it is foreseeable that the workload will double within the next month. Using workload forecasting approaches developed in [18, 19], it is possible to forecast future workload intensity trends. Based on the latter, our approach allows performance engineers and system administrators to anticipate performance problems. System behavior and performance can be easily evaluated for different workloads. In contrast to performance tests, the model-based approach allows evaluating the system without setting up a representative testbed. The predictions allow both determining the maximal system throughput as well as detecting potential bottlenecks. The questions that arise in this scenario are:

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

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

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

Autonomic Resource Management at Run-time DML provides a basis for developing model-based *autonomic* performance and resource management techniques that proactively adapt the system to dynamic changes at run-time with the goal to satisfy performance objectives while at the same time ensuring efficient resource utilization.

State-of-the-art industrial mechanisms for automated performance and resource management generally follow a trigger-based approach when it comes to enforcing application-level Service Level Agreements (SLAs) concerning availability or responsiveness. Custom triggers can be configured that fire in a reactive manner when an observed metric reaches a certain threshold (e.g., high server utilization or long service response times) and execute certain predefined reconfiguration actions until a given stopping criterion is fulfilled (e.g., response times drop) [20, 21]. However, application-level metrics, such as availability and responsiveness, normally exhibit a highly non-linear behavior on system load and they typically depend on the behavior of multiple servers across several application tiers. Hence, it is hard to determine general thresholds of when triggers should be fired given that the appropriate triggering points are typically highly dependent on the architecture of the hosted services and their workload profiles, which can change frequently during operation. The inability to anticipate and predict the effect of dynamic changes in the environment, as well as to predict the effect of possible adaptation actions, renders conventional trigger-based approaches unable to reliably enforce SLAs in an efficient and proactive fashion.

To overcome the mentioned shortcomings of current industrial approaches, [22] developed a framework for autonomic performance-aware resource management. Figure 1.3 shows the control loop that is central to that framework. It consists of four main phases *Monitor*, *Analyze*, *Plan* and *Execute*. In

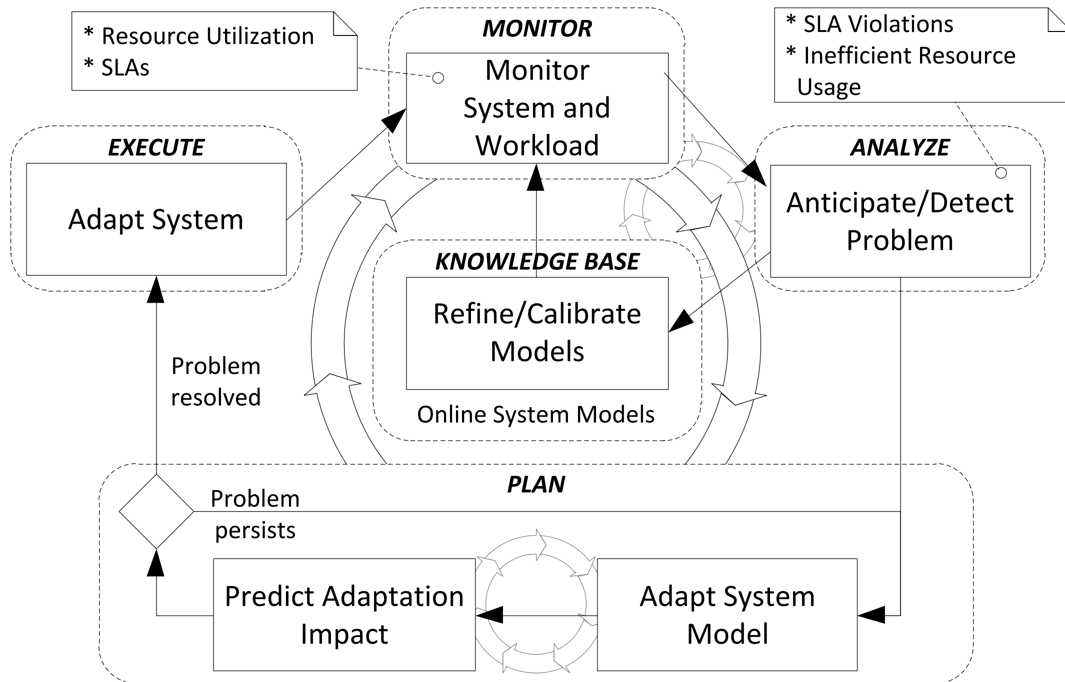


Figure 1.3: Model-Based System Adaptation Control Loop [22]

addition, the figure depicts a *Knowledge Base* that is used by all mentioned phases. The knowledge base is realized with DML. DML is used to conduct performance predictions on the model level to anticipate performance problems and to find suitable adaptation actions. Given that DML supports detailed impact analyses, e.g., workload intensity and usage profile changes, service (re-)compositions or deployment changes, the adaptation mechanisms can quickly converge to an efficient target system configuration [23]. The tailored prediction process allows the adaptation mechanism to trigger predictions for multiple different configuration scenarios within a controllable period of time. The prediction results are sufficiently accurate since the models are maintained up-to-date based on representative monitoring data obtained at run-time.

[22] evaluates the framework end-to-end in two different representative case studies, demonstrating that it can provide significant efficiency gains of up to 50% without sacrificing performance guarantees, and that it is able to trade-off performance requirements of different customers in heterogeneous hardware environments. Furthermore, it is shown that the approach enables proactive system adaptation, reducing the amount of SLA violations by 60% compared to a trigger-based approach. The results of the case studies in [22] show that it is possible to apply architecture-level performance models and online performance prediction to perform autonomic system adaptation on the model level such that the system's operational goals are maintained. Different adaptation possibilities can be assessed without having to change the actual system.

1.4 Self-Aware Computing Systems

As mentioned above, a major application of the Descartes Modeling Language (DML) is to serve as a basis for *self-aware* resource management during operation. Self-aware computing systems are best understood as a sub-class of autonomic computing systems. In this section, we explain in more detail what exactly is meant by *self-awareness* in this context.

DML is a major part of our broader long-term research effort[2] aimed at developing novel methods, techniques and tools for the engineering of *self-aware* computing systems [1, 2]. The latter are designed with built-in online QoS prediction and self-adaptation capabilities used to enforce QoS requirements in a cost- and energy-efficient manner. Self-awareness in this context is defined by the combination of three properties that a system should possess:

1. *Self-reflective*: Aware of its software architecture, execution environment, and hardware infrastructure on which it is running as well as of its operational goals (e.g., QoS requirements, cost- and energy-efficiency targets),
2. *Self-predictive*: 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),
3. *Self-adaptive*: Proactively adapting as the environment evolves in order to ensure that its operational goals are continuously met.

The Descartes Modeling Language (DML) is designed with the goal to provide modeling abstractions to capture and express the system architecture aspects whose knowledge is required at run-time to realize the above three properties. A major goal of these abstractions is to provide a balance between model expressiveness, flexibility and compactness. Instances of the various parts of the meta-model are intended to serve as *online models* integrated into the system components they represent and reflecting all aspects relevant to managing their QoS and resource efficiency during operation.

In parallel to this, we are working on novel application platforms designed to automatically maintain online models during operation to reflect the evolving system environment. The online models are intended to serve as a “mind” to the running system controlling its behavior at run-time, i.e., deployment configurations, resource allocations and scheduling decisions. To facilitate the initial model construction and continuous maintenance during operation, we are working on techniques for automatic model extraction based on monitoring data collected at run-time [24, 25, 26].

The online system models make it possible to answer QoS-related queries during operation such as for example: What would be the effect on the QoS of running applications and on the resource consumption of the infrastructure if a new service is deployed in the virtualized environment or an existing service is migrated from one server to another? How much resources need to be allocated to a newly deployed service to ensure that SLAs are satisfied while maximizing energy efficiency? What QoS would a service exhibit after a period of time if the workload continues to develop according to the current trends? How should the system configuration be adapted to avoid QoS problems or inefficient resource usage arising from changing customer workloads? What operating costs does a service hosted on the infrastructure incur and how does the service workload and usage profile impact the costs? We refer to such queries as *online QoS queries*.

The ability to answer online QoS queries during operation provides the basis for implementing techniques for self-aware QoS and resource management. Such techniques are triggered automatically during operation in response to observed or forecast changes in the environment (e.g., varying application work-

[2] http://www.descartes-research.net

loads). The goal is to *proactively* adapt the system to such changes in order to avoid anticipated QoS problems and/or inefficient resource usage. The adaptation is performed in an autonomic fashion by considering a set of possible system reconfiguration scenarios (e.g, changing virtual machine placement and/or changing resource allocations) and exploiting the online QoS query mechanism to predict the effect of such reconfigurations before making a decision [27]. Each time an online QoS query is executed, it is processed based on the online system architecture models (DML instances) provided on demand by the respective system components during operation. Given the wide range of possible contexts in which the online models can be used, automatic model-to-model transformation techniques (e.g., [28]) are used to generate tailored prediction models on-the-fly depending on the required accuracy and the time available for the analysis. Multiple prediction model types and model solution techniques are employed here in order to provide flexibility in trading-off between prediction accuracy and analysis overhead.

1.5 Outline

The remainder of this technical report is organized as follows. In Chapter 2, we provide an overview on related work concerning performance modeling on the one hand and run-time system reconfiguration and adaptation on the other hand. Chapter 3 introduces a representative online prediction scenario we use throughout the technical report to motivate and evaluate the novel modeling approaches. The application architecture and resource landscape models, i.e., the system architecture QoS model is described in Chapter 4. Our approach to modeling system adaptation is presented in Chapter 5. The report concludes with a discussion of the differences between DML and PCM, and provides an outlook on future work in Chapter 6.

Chapter 2

Background

In this chapter, we provide a brief overview of the state-of-the-art on performance modeling and prediction (Section 2.1), on the one hand, and the state-of-the-art on run-time system adaptation (Section 2.2), on the other hand.

2.1 Performance Modeling Approaches

We first present an overview of current performance modeling approaches for IT systems focusing on architecture-level performance models. We then introduce the Palladio Component Model (PCM) [11], a meta-model for design-time performance analysis of component-based software architectures, which has inspired some of the core elements of the Descartes Modeling Language (DML).

A survey of model-based performance prediction techniques was published in [29]. A number of techniques utilizing a range of different performance models have been proposed including product-form queueing networks (e.g., [7]), layered queueing networks (e.g., [9]), queueing Petri nets (e.g., [30]), stochastic process algebras [31], statistical regression models (e.g., [32]) and learning-based approaches (e.g., [33]). Such models capture the temporal system behavior and can be used for performance prediction by means of analytical or simulation techniques. We refer to them as *predictive* performance models.

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

2.1.1 Existing Architecture-Level Performance Models

Architecture-level[1] performance models provide means to model the performance-relevant aspects of system architectures at a more detailed level of abstraction. Such models are *descriptive* in nature (e.g., software architecture models based on UML, annotated with descriptions of the system's performance-relevant behavior) and they can normally be transformed automatically into predictive performance

[1] Architecture-level in this context is meant in a broader sense covering both the system's software architecture and execution environment.

models allowing to predict the system performance for a given workload and configuration scenario. Architecture-level performance models 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.

Over the past decade, a number of architecture-level performance meta-models have been developed by the performance engineering community. The most prominent examples are the UML SPT profile [34] and its successor the UML MARTE profile [14], both of which are extensions of UML as the de facto standard modeling language for software architectures. Other proposed meta-models include CSM [35], PCM [11], SPE-MM [13], and KLAPER [12]. A recent survey of model-based performance modeling techniques for component-based systems was published in [15].

While architecture-level performance models provide a powerful tool for performance prediction, they are typically expensive to build and provide limited support for reusability, parameterization and customization, which renders them impractical for use in online scenarios. Recent efforts in the area of component-based performance engineering [15] have contributed a lot to facilitate model reusability, however, given that such models are designed for offline use at system design time, they assume a *static* system architecture and do not support modeling dynamic system aspects [36]. In a modern virtualized system environment dynamic changes are common, e.g., service workloads change over time, new services are deployed, or virtual machines are migrated between servers. The amount of effort involved in maintaining performance models is prohibitive and therefore, in practice, such models are rarely used after deployment [37].

2.1.2 Palladio Component Model (PCM)

One of the more advanced architecture-level performance modeling languages, in terms of parametrization and tool support, is the Palladio Component Model (PCM) [11]. PCM is a meta-model designed to support the analysis of quality attributes (performance, reliability and maintainability) of component-based software architectures. It is targeted at design-time performance analysis, i.e., enabling performance predictions early in the development lifecycle to evaluate different system design alternatives. In the following, we present a brief overview of PCM since it was used as a basis for some core elements of DML.

In PCM, the component execution context is parameterized to explicitly capture the influence of the component's connections to other components, its allocated hardware and software resources, and its usage profile including service input parameters. Model artifacts are divided among the developer roles involved in the component-based software engineering process, i.e., component developers, system architects, system deployers and domain experts.

PCM models are divided into five sub-models:

- *Component models*, stored in a component repository, which describe the performance relevant aspects of software components, i.e., control flow, resource demands, parameter dependencies, etc.
- *System model* which describes how component instances from the repository are assembled to build a specific system.
- *Resource environment model* which specifies the execution environment in which the system is deployed.
- *Allocation model* which describes what resources from the resource environment are allocated to the components defined in the system model.
- *Usage model* which describes the user behavior, i.e., the services that are called, the frequency and

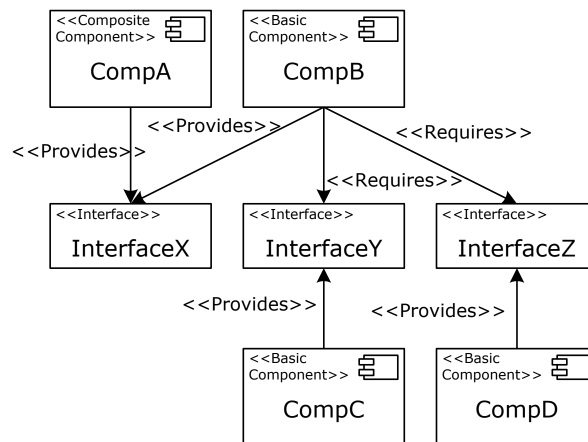


Figure 2.1: Components providing and requiring interfaces.

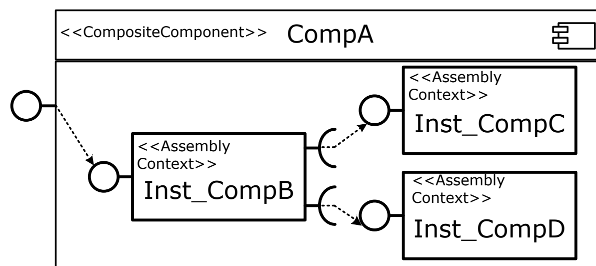


Figure 2.2: Assembly of a composite component.

order in which they are invoked, and the input parameters passed to them.

Component Model and System Model In PCM, a component repository contains component and interface specifications, i.e., interfaces are explicitly modeled. Components and interfaces are connected using so-called roles: A component specification consists of a list of provided roles (referring to interfaces the component provides) and a list of required roles (referring to interfaces the component requires). An interface is specified as a set of method signatures.

A component may be either a *basic component* or a *composite component*. Figure 2.1 shows an example illustrating basic components, composite components and interfaces. Composite component *CompA* and basic component *CompB* both provide interface *InterfaceX*. The interfaces required by component *CompB* are provided by *CompC* and *CompD*, respectively.

A composite component may contain several child component instances assembled through so-called *assembly connectors* connecting required interfaces with provided interfaces. Connectors from the child component instances to the composite component boundary are modeled using delegation connectors. Each service the composite component provides and each service it requires has to be linked to a child component instance using such delegation connectors. A component-based *system* is modeled as a designated composite component that provides at least one interface. An example of how a composite component is assembled is shown in Figure 2.2. Component *CompA* comprises three instances of basic components introduced in Figure 2.1 connected according to their provided and required interfaces.

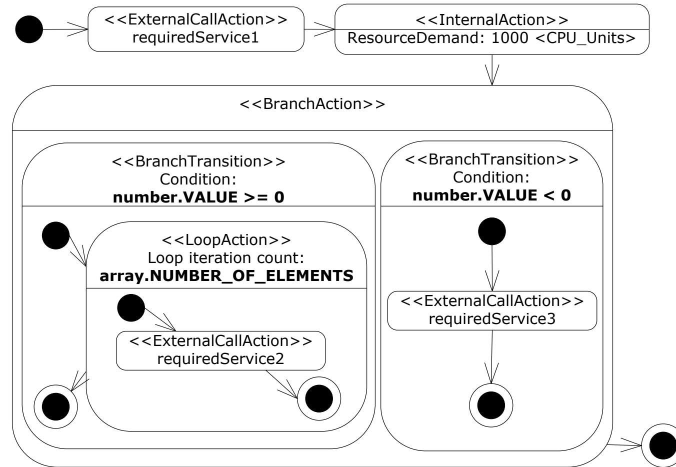


Figure 2.3: RD-SEFF of service with signature *execute(int number, List array)* (cf. [11]).

Service Behavior Abstraction For each service a component provides, in PCM the service's internal behavior is modeled using a Resource Demanding Service Effect Specification (RDSEFF). An RDSEFF captures the control flow and resource consumption of the service depending on its input parameters passed upon invocation. The control flow is abstracted covering only performance-relevant actions. Calls to required services are modeled using so-called ExternalCallActions, whereas internal computations within the component are modeled using InternalActions. Control flow actions like LoopAction or BranchAction are only used when they affect calls to required services, e.g., if a required service is called within a loop; otherwise, a loop is captured as part of an InternalAction. LoopActions and BranchActions can be characterized with loop iteration numbers and branch probabilities, respectively. An example of an RDSEFF for the service *execute(int number, List array)* [11] is shown in Figure 2.3. It is depicted in a notation similar to UML activity diagrams. First, a required service is invoked using an ExternalCallAction and then an InternalAction is executed. Following this, there is a BranchAction with two BranchTransitions. The first BranchTransition contains a LoopAction whose body consists of another ExternalCallAction. The second BranchTransition contains a further ExternalCallAction.

The performance-relevant behavior of the service is parameterized with service input parameters. Whether the first or second BranchTransition is called depends on the value of service input parameter *number*. This parameter dependency is specified *explicitly* as a branching condition. Similarly, the loop iteration count of the LoopAction is modeled to be equal to the number of elements of the input parameter *array*. PCM also allows to define parameter dependencies stochastically, i.e., the distribution of the loop iteration count can be described with a probability mass function (PMF): `IntPMF[(9;0.2) (10;0.5) (11;0.3)]`. The loop body is executed 9 times with a probability of 20%, 10 times with a probability of 50%, and 11 times with a probability of 30%. Note that this probabilistic description remains component type-specific, i.e., it should be valid for all instances of the component.

In PCM, performance behavior abstractions are encapsulated in the component type specifications enabling performance predictions of component compositions at design-time. However, as we show in the next section, such design-time abstractions are not suitable for use in online performance models due to the limited flexibility in expressing and resolving parameter and context dependencies at run-time. Furthermore, we show that in many practical situations, providing an *explicit* specification of a parameter

dependency as discussed above is not feasible and an empirical representation based on monitoring data is more appropriate.

Usage Model A usage model represents the usage profile (also called workload profile) of the modeled system. It describes which services are called and what input parameters are passed to them. In addition, the order in which the services are called as well as the workload intensity can be specified. In this way, the user behavior and the resulting the system workload can be described.

Mapping of Software Component Instances to Resources To complete the performance model, the modeled software component instances have to be mapped to resources in a so-called *allocation model*. The component instances are defined in the system model, whereas the available resources are defined in the *resource environment model*. In PCM, a resource environment model consists of resource containers representing hardware servers. A resource container contains processing resources such as CPUs and storage devices.

2.2 Modeling Run-time System Adaptation

This section discusses the state-of-the-art related to DML's adaptation points model and the adaptation language which are presented in detail in Chapter 5. More specifically, we contrast the abstraction levels employed in existing approaches to system adaptation and we briefly review other languages for system adaptation as well as alternative approaches to defining adaptation points in architecture models.

2.2.1 Abstraction Levels

Architectural models provide common means to abstract from the system details and analyze system properties. Such models have been used for self-adaptive software before, e.g., in [38, 39], however, existing approaches do not explicitly capture the degrees of freedom of the system configuration as part of the models. Other approaches use a three-level abstraction of the adaptation processes, e.g., in [40] to specify policy types for autonomic computing or especially in [41], defining an ontology of tactics, strategies and operations to describe self-adaptation. However, to the best of our knowledge, none of the existing approaches separates the specification of the models at the three levels, explicitly distinguishing between different system developer roles. By separating the knowledge about the adaptation process and encapsulating it in different sub-models, we can reuse this knowledge in other self-adaptive systems or reconfiguration processes.

2.2.2 Languages for Adaptation Control Flow

In [42], Cheng introduces Stitch, a programming language-like notation for using strategies and tactics to adapt a given system. 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. Other languages like Service Activity Schemas (SAS) [43] or the Business Process Execution Language (BPEL) [44]) are very application specific and also describe adaptation processes with pre-defined control flows. Moreover, because of their focus on modeling business processes, these approaches are not able to model the full spectrum of self-adaptive mechanisms from conditional expressions to algorithms and heuristics as presented by [38].

2.2.3 Configuration Space

In the area of automated software architecture improvement, most existing approaches use a fixed representation of the configuration space and thus do not allow to freely model a configuration space. Two notable exceptions are PETUT-MOO and the Generic Design Space Exploration Framework (GDSE). The PETUT-MOO approach [45] uses model transformations to describe changes in the configuration of software architectures. However, this idea has not been followed up in later works of the authors, which focuses on architecture optimization and does not describe the configuration space in detail.

Saxena et al. [46] have presented a self-adaptation approach using the GDSE framework. The configuration space is represented as an AND-OR-tree describing possible design options and their dependencies. The quality effects of such options are directly encoded in the tree. As a result, the quality functions to consider are limited to arithmetic expressions on architecture properties (such as "the sum or component latencies make up the overall latency of an embedded system") and an arbitrary quality evaluation of the architecture (e.g., using stochastic models) is not supported.

A closely related approach to modeling the configuration space of a software architecture is PCM's Degree-of-Freedom Meta-Model [47] allowing to capture different types of configuration changes—such as changes to add vCPUs, to add servers, and to exchange software components—in a single configuration model. However, this Degree-of-Freedom Meta-Model describes the configuration possibilities on the meta-model level, i.e., all instances of this meta-model have the same variability. In contrast, in the context of DML, we describe configuration possibilities on the model instance level because each modeled system has its own configuration possibilities.

Chapter 3

Online Performance Prediction Scenario

The scenario presented in this chapter serves as a concrete example to motivate and illustrate the concepts and goals of the Descartes Modeling Language (DML). Moreover, it gives an overview of the foundations and technical background this work builds on. An implementation of this scenario serves as a reference system to evaluate the new modeling concepts.

3.1 Setting

We consider a scenario where a set of customers are running their applications in a virtualized data center infrastructure. Each customer is assigned one application server cluster. A shared database is deployed on a centralized server. Each customer can have different performance objectives, i.e., Service Level Agreements (SLAs), that have to be enforced by the service provider. As part of the customer SLAs, the expected service workloads for which SLAs are established must be specified.

We assume that each customer has their own independent workload and that the workload intensity can vary over time. As a first step, we assume that the customer will provide the service provider with information about expected workload changes in advance (e.g., expected increase in the workload due to a planned sales promotion). In addition, we are currently working on integrating automatic workload forecasting mechanisms. The challenge is the how to proactively adapt the system to workload changes in order to ensure that customer SLAs are continuously satisfied while utilizing system resources efficiently. This includes the anticipation of workload changes and the triggering of corresponding system reconfigurations. As an example of a realistic and representative enterprise application, we employ the SPECjEnterprise2010 standard benchmark.

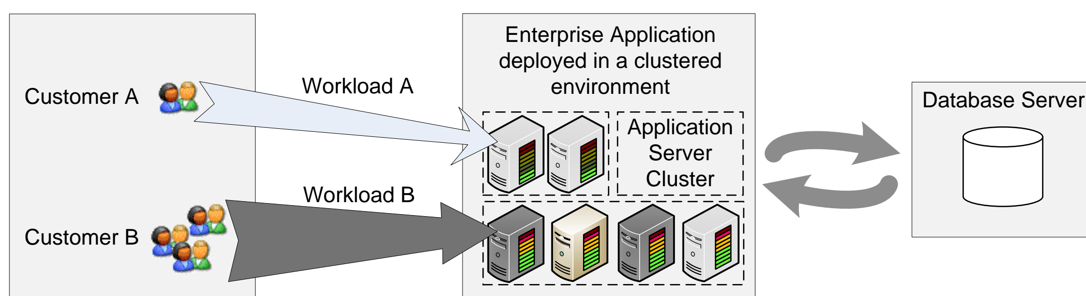


Figure 3.1: Online performance prediction scenario.

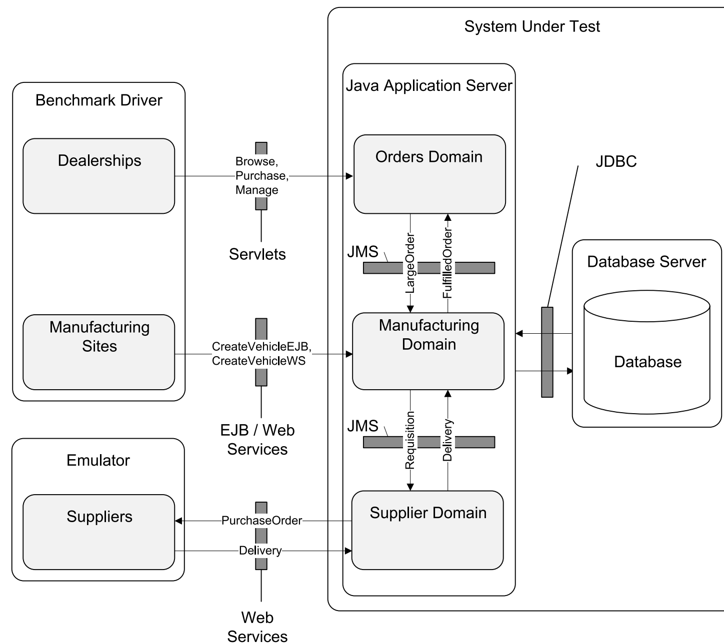


Figure 3.2: SPECjEnterprise2010 architecture [48].

3.2 SPECjEnterprise2010

SPECjEnterprise2010[1] is a Java EE benchmark developed by SPEC's Java Subcommittee for evaluating the performance and scalability of Java EE-based application servers. It implements a business information system of a representative size and complexity. The benchmark workload is generated by an application that is modeled after an automobile manufacturer. As business scenarios, the application comprises customer relationship management (CRM), manufacturing, and supply chain management (SCM). The business logic is divided into three domains: orders domain, manufacturing domain and supplier domain.

To give an example of the business logic implemented by the benchmark, consider a car dealer that places a large order with the automobile manufacturer. The large order is sent to the manufacturing domain which schedules a work order to manufacture the ordered vehicles. In case some parts needed for the production of the vehicles are depleted, a request to order new parts is sent to the supplier domain. The supplier domain then selects a supplier and places a purchase order. When the ordered parts are delivered, the supplier domain contacts the manufacturing domain and the inventory is updated. Finally, upon completion of the work order, the orders domain is notified.

Figure 3.2 depicts the architecture of the benchmark as described in the benchmark documentation. The benchmark application is divided into three domains, aligned to the business logic: orders domain, manufacturing domain and supplier domain. In the three domains, the application logic is implemented using Enterprise Java Beans (EJBs) which are deployed on the considered Java EE application server. The domains interact with a database server via Java Database Connectivity (JDBC) using the Java Persistence API (JPA). The communication between the domains is asynchronous and implemented using point-to-point messaging provided by the Java Message Service (JMS). The workload of the orders domain is triggered by dealerships whereas the workload of the manufacturing domain is triggered by

[1] http://www.spec.org/jEnterprise2010/

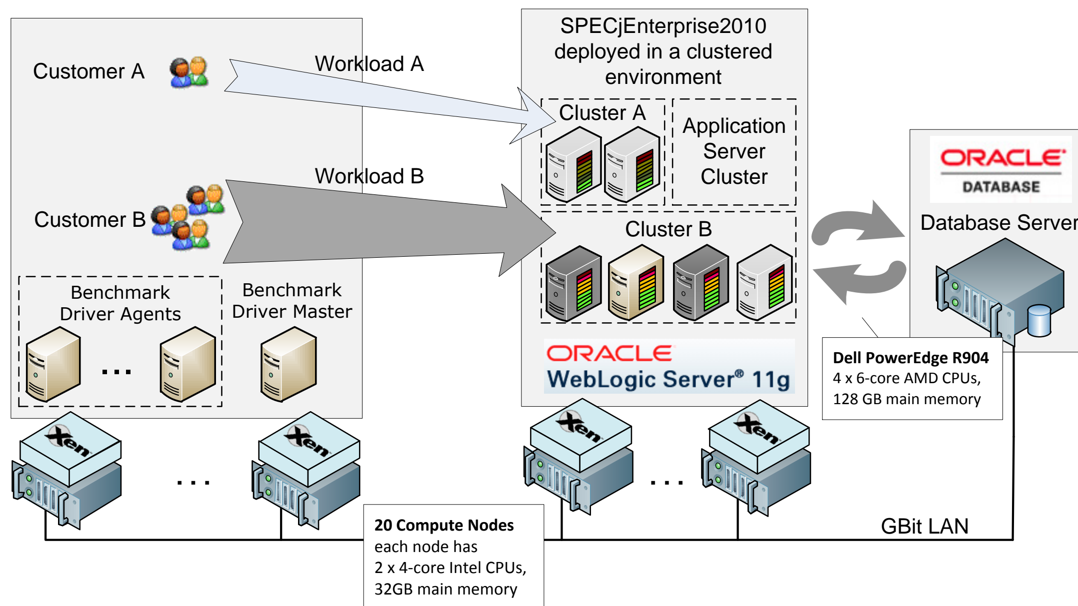


Figure 3.3: Experimental environment.

manufacturing sites. Both, dealerships and manufacturing sites are emulated by the benchmark driver, a separate supplier emulator is used to emulate external suppliers. The communication with the suppliers is implemented using Web Services. While the orders domain is accessed through Java Servlets, the manufacturing domain can be accessed either through Web Services or EJB calls, i.e., Remote Method Invocation (RMI).

As shown on the diagram, the system under test (SUT) spans both the Java application server and the database server. The emulator and the benchmark driver have to run outside the system under test so that they do not affect the benchmark results. The benchmark driver executes five benchmark operations. A dealer may `browse` through the catalog of cars, `purchase` cars, or `manage` his dealership inventory, i.e., sell cars or cancel orders. In the manufacturing domain, work orders for manufacturing vehicles are placed, triggered either through WebService or RMI calls (`createVehicleWS` or `createVehicleEJB`).

3.3 Exemplary Environment

We implemented the described scenario in the experimental environment depicted in Figure 3.3. As virtualization layer, we used the Xen Cloud Platform[2], an open source infrastructure platform that provides standard resource management functionality as well as additional features such as high availability or management facilities based on standardized APIs. It is based on the Xen hypervisor by Citrix which is one of the major virtualization platforms used in industry. For the deployment of the application server tier, we used Oracle WebLogic Server (WLS) 10.3.3 instances. Each WLS instance runs on a machine with 2x4-core Intel CPUs with OpenSuse 11.1. As database server (DBS), we used Oracle Database 11g, running on a 24-core Dell PowerEdge R904. The benchmark driver and the supplier emulator were

[2]The Xen Cloud Platform, http://www.xen.org/products/cloudxen.html

running on virtualized blade servers. The machines are connected by a 1 GBit LAN. The presented environment can be considered as representative of a modern business information system.

For each customer in our scenario, a separate instance of the benchmark is deployed in one application server cluster assigned to the respective customer. The customer's workload is generated by a customized instance of the benchmark driver. The operations executed by the SPECjEnterprise2010 benchmark are `Browse`, `Purchase`, `Manage`, `CreateVehicleEJB` and `CreateVehicleWS`. As an example of an SLA, the customer could require that the response time of `Purchase` must not exceed $5ms$ or, less restrictive, must be below $5ms$ in 95% of the cases within a given time horizon (e.g., one hour).

Chapter 4

Architecture-Level Performance Model

In this chapter, we present the application architecture and resource landscape sub-meta-models of Descartes Modeling Language (DML), i.e., collectively used to define a *system architecture QoS model*. The current version of DML is focused on performance including capacity, responsiveness and resource efficiency aspects, however, work is underway to provide support for modeling further QoS properties. The meta-model itself is designed in a generic fashion and is intended to eventually support the full spectrum of QoS properties mentioned in Section 1. Given that the initial version of DML is focussed on performance, in the rest of this chapter, we mostly speak of performance instead of QoS in general.

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

4.1 Application Architecture Model

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

The application architecture meta-model is described in several subsections. Subsection 4.1.1 describes the underlying component model. Subsection 4.1.2 introduces a running example that is used to motivate and illustrate the new modeling concepts. Subsection 4.1.3 introduces novel service behavior abstractions. In Subsection 4.1.4, we present how the behavior abstractions are parameterized and in Subsection 4.1.5 we describe how we model probabilistic parameter dependencies specifically for use at run-time. An interface used to obtain empirical characterizations of model parameters from monitoring statistics is described in Subsection 4.1.6.

4.1.1 Component Model and System Model

The component model stems from the Palladio Component Model (PCM) [11, 49]. Software building blocks are modeled as components. In the following, we describe how components are associated with interfaces they provide or require, and how composite components can be assembled from other components.

Components and interfaces are modeled as separate model entities, i.e., components as well as interfaces are first-class entities that can exist on their own. Consequently, a component does not *contain* an interface, but it may provide and/or require some interfaces [17]. The connection between components and interfaces is specified using so-called *roles* [11]. A component can take two roles relative to an interface. It can either provide and implement the functionality specified in the interface or it can require that functionality. Figure 4.1 shows the corresponding meta-model. An InterfaceProvidingEntity may

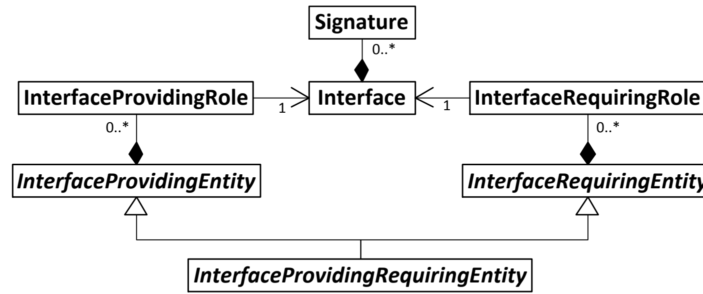


Figure 4.1: Components and Interfaces, cf. [11]

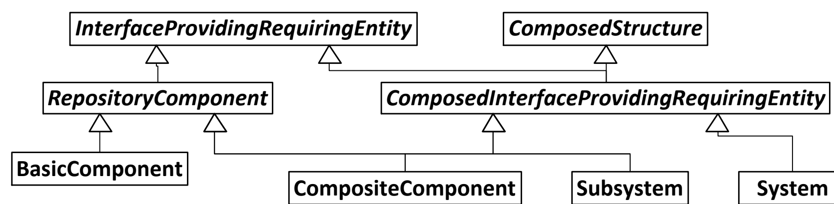


Figure 4.2: Component Type Hierarchy, cf. [11]

have InterfaceProvidingRoles that refer to an Interface consisting of one or more method Signatures. An InterfaceRequiringEntity is modeled accordingly with InterfaceRequiringRoles. We refer to each method provided by a component as a service. We thus refer to the methods of the provided interfaces of a component as provided services, and refer to the methods of the required interfaces of a component as required services or external services.

An InterfaceProvidingRequiringEntity is the supertype of different component types that are shown in Figure 4.2. Basically, two types of components are distinguished. BasicComponents, i.e., atomic components, and CompositeComponents both can require and provide interfaces, and are stored in a component repository. Thus, they are subtypes of InterfaceProvidingRequiringEntity and RepositoryComponent. A CompositeComponent also inherits from type ComposedStructure, indicating that it is composed of other components. A Subsystem is similar to a CompositeComponent, but treated differently when it comes to modeling the deployment of components. While a CompositeComponent is deployed as a whole, the Subsystem is deployed by deploying all its child components. A System is similar to a CompositeComponent, the difference is that a System is not part of a component repository but a unique designated ComposedStructure. A System is the outermost ComposedStructure representing the system boundary.

Figure 4.3 shows how a ComposedStructure is assembled. A ComposedStructure may contain several AssemblyContexts which themselves each refer to a RepositoryComponent (referring to a Subsystem is only allowed if the parent ComposedStructure is of type System or Subsystem). Each AssemblyContext thus represents a child component instance in the composite. An AssemblyConnector connects two such child component instances with an InterfaceRequiringRole and an InterfaceProvidingRole, representing a connection between a providing role of the first component and a requiring role of the second component. Connectors from a child component instance to the composite component boundary are modeled using delegation connectors (InterfaceProvidingDelegationConnector and InterfaceRequiringDelegationConnector). The delegation connectors refer to a role of an inner child component instance and to a role of

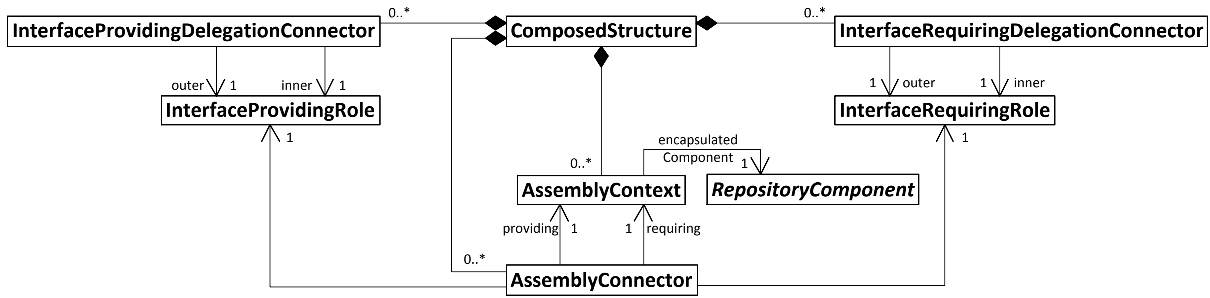


Figure 4.3: Component Composition, cf. [11]

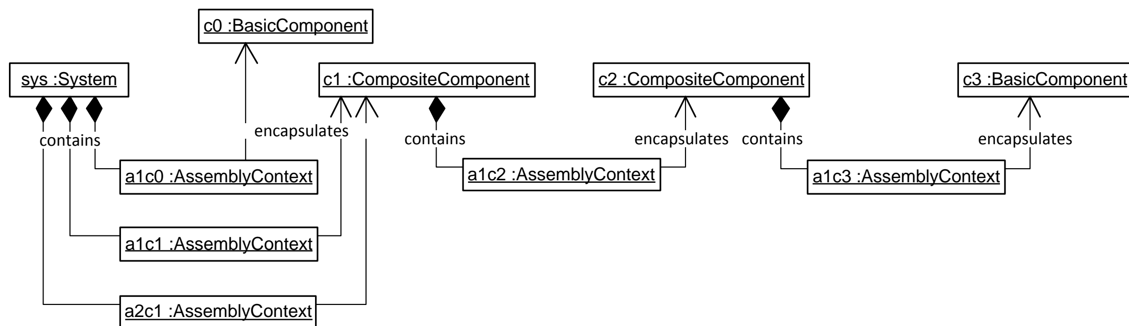


Figure 4.4: Example: System Instance as UML Object Diagram

the outer ComposedStructure.

Figure 4.4 shows an exemplary instantiation of a ComposedStructure. It shows a System model as UML object diagram. The system instance *sys* contains three AssemblyContexts. Two of them refer to the same CompositeComponent *c1*, one refers to BasicComponent *c0*. Component *c1* contains an AssemblyContext *a1c2* that refers to another CompositeComponent *c2* that itself encapsulates BasicComponent *c3* via AssemblyContext *a1c3*. Figure 4.5 shows the same instance as component diagram. The outermost box represents System *sys*. AssemblyContext *a1c0* is connected to AssemblyContexts *a1c1* and *a2c1*, e.g., it could balance the load between the two instances of CompositeComponent *c1*.

Although there are only one AssemblyContext for component *c2* and only one AssemblyContext for component *c3*, the system diagram in Figure 4.5 illustrates that both components must be instantiated twice, because there are two instances of their surrounding component *c1*. Thus, an AssemblyContext is not equivalent to a component instance. An AssemblyContext is only unambiguous within its direct parent composite structure.

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

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

- Each tree node $v \in V$ represents a component instance, denoted as $instance(v)$.
- $\forall v \in V$:

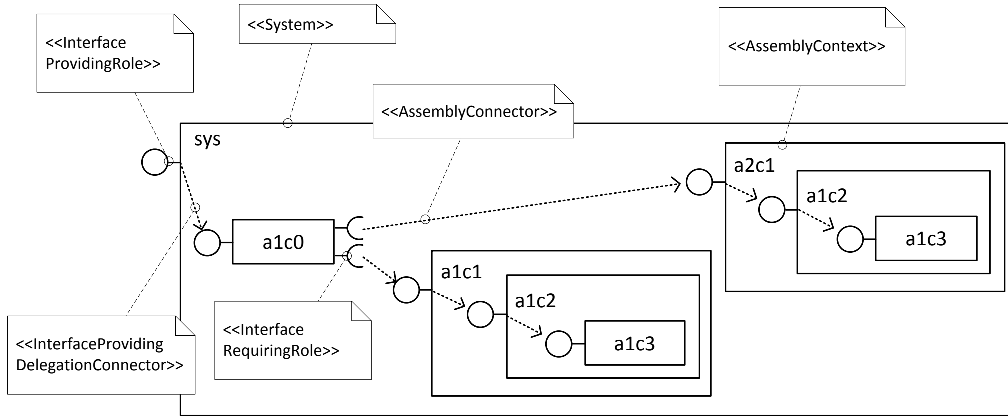


Figure 4.5: Example: System Instance

$(\exists e = (v, v') \in E$
$\Longleftrightarrow$
$instance(v)$ has a child AssemblyContext that encapsulates $instance(v'))$

The inner nodes of G represent instances of ComposedStructures, the leaves of G represent instances of BasicComponents. Figure 4.6(a) illustrates such a composition tree.

In a ComposedStructure *cs*, a component instance is uniquely identified by a path from the node representing *cs* to the node representing the specific component instance. Figure 4.6(b) shows the example of Figure 4.4 as composition tree. The root of the tree is System *sys*. The two instantiations of CompositeComponent *c1* are highlighted with dashed lines. In the example, the two instances of BasicComponent *c3* can thus be identified by the path {*a1c1*, *a1c2*, *a1c3*} and the path {*a2c1*, *a1c2*, *a1c3*}, respectively.

A ComponentInstanceReference can thus be modeled as a sequence of AssemblyContexts as shown in Figure 4.7. The difference between a component type and its instances is of relevance in Section 4.1.4 where we distinguish between parameterizations at the component type level and at the component instance level.

4.1.2 Running Example

We introduce a running example to illustrate the novel modeling abstractions we propose in the following sections. Figure 4.8 shows a simple online shop, consisting of a *WebShop* composite component, and an *SQLDB* component. The *WebShop* consists of several Java Servlet components, the entity data is accessed using a Java Persistence API (JPA) provider component (*JPAProvider*). The *CatalogServlet* allows browsing the catalog of available articles. The *ShowDetailsServlet* implements a view of the article details. The *ShoppingCartServlet* provides a shopping cart including payment processing and requires an external *ArticleDelivery* service that is implemented by the *Delivery* component.

4.1.3 Service Behavior Abstractions

This section introduces different service behavior abstraction levels. Section 4.1.3.1 provides the motivation for the new service behavior abstractions, Section 4.1.3.2 describes the modeling approach and

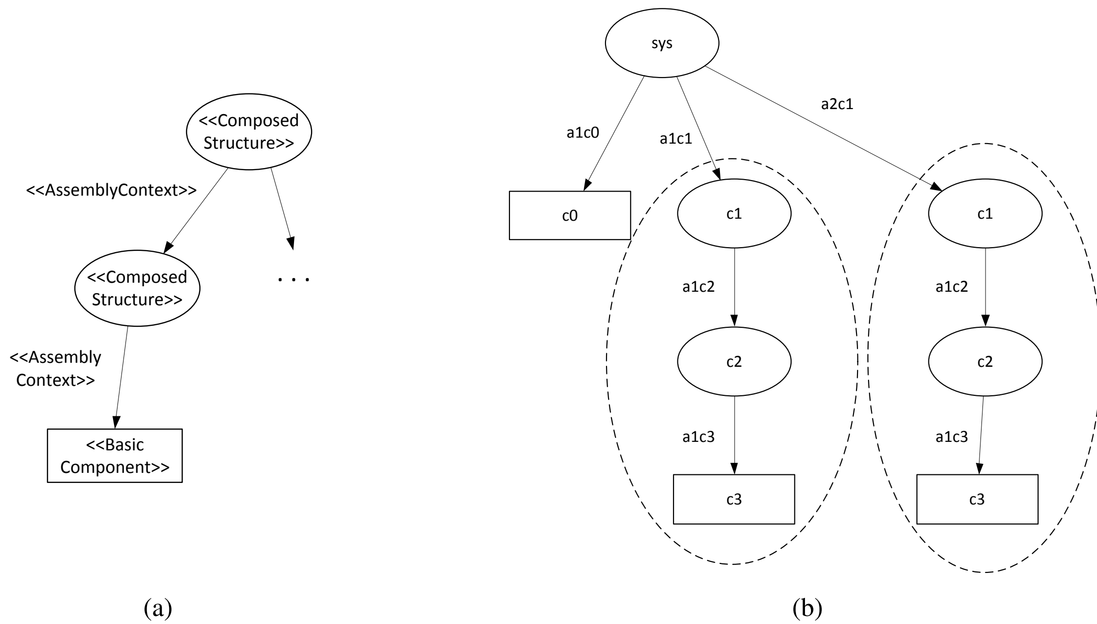


Figure 4.6: (a) Composition Tree Schema and (b) Example System Instance as Composition Tree

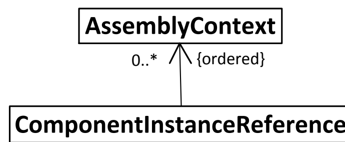


Figure 4.7: Component Instance Reference

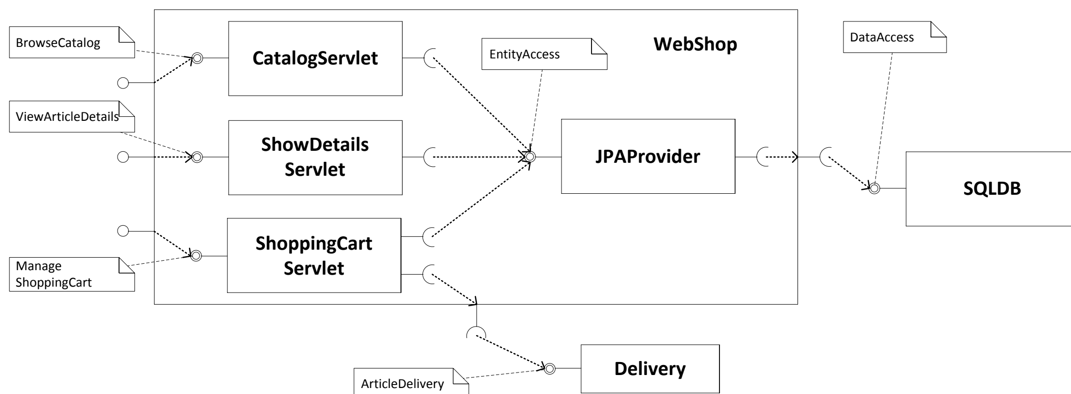


Figure 4.8: Running Example: WebShop

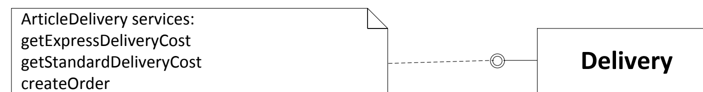


Figure 4.9: Example: *Delivery* Component

Section 4.1.3.3 describes the corresponding meta-model in detail. In Section 4.1.3.4, we provide an illustrative example.

4.1.3.1 Motivation

In order to ensure Service Level Agreements (SLAs) while at the same time optimizing resource utilization, the service provider needs to be able to predict the system performance under varying workloads and dynamic system reconfigurations. The underlying performance models enabling online performance prediction must be parameterized and analyzed on-the-fly. Such models may be used in many different scenarios with different requirements for accuracy and timing constraints. Depending on the time horizon for which a prediction is made, online models may have to be solved within seconds, minutes, hours, or days, and the same model should be usable in multiple different scenarios with different requirements for prediction accuracy and analysis overhead. Hence, in order to provide flexibility at run-time, our meta-model must be designed to support multiple abstraction levels and different analysis techniques allowing to trade-off between prediction accuracy and time-to-result.

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

In the following, we describe three practical examples where models at different abstraction levels are needed, based on the *WebShop* example introduced in Figure 4.8. The *Delivery* component provides services to calculate the cost of a delivery and to create a delivery order (see Figure 4.9). Two kinds of deliveries are supported: a standard delivery and an express delivery.

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

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

Finally, if the internal behavior of the *Delivery* component including its control flow and resource consumption of internal actions can be monitored, more detailed models can be built allowing to obtain

more accurate performance predictions including response time distributions. Predicting response time distributions is relevant for example in situations where SLAs with service response time limits defined in terms of response time percentiles need to be evaluated.

In summary, it is important to support the modeling of service behavior at different levels of abstraction and detail. The models should be usable in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to accurate performance prediction. Furthermore, the modeled abstraction level depends on the information that monitoring tools can obtain at run-time, e.g., to what extent component-internal information is available.

4.1.3.2 Modeling Approach

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

- **Black-box behavior abstraction.** A “black-box" abstraction is a probabilistic representation of the service response time behavior. Resource demands are not specified. This representation captures the view of the service behavior from the perspective of a service consumer without any additional information about the service’s behavior.

- **Coarse-grained behavior abstraction.** A “coarse-grained" abstraction captures the service behavior when observed from the outside at the component’s boundaries. It consists of a description of the frequency of external service calls and the overall service resource demands. Information about the service’s total resource consumption and information about external calls made by the service is required, however, no information about the service’s internal control flow is assumed.

- **Fine-grained behavior abstraction.** A “fine-grained" abstraction captures the performance-relevant service control flow which is an abstraction of the actual control flow. Performance-relevant actions are component-internal computational tasks, the acquisition and release of locks, as well as external service calls, thus also loops and branches where external services are called. Furthermore, the ordering of external service calls and internal computations may have an influence on the service performance. The control flow is modeled at the same abstraction level as the Resource Demanding Service Effect Specification (RDSEFF) of PCM (cf. [11]), however, there are significant differences in the way model variables and parameter dependencies are modeled. The details of these are presented in Section 4.1.4 and Section 4.1.5. In contrast to the coarse-grained behavior description, a fine-grained behavior description requires information about the internal performance-relevant service control flow including information about the resource consumption of internal service actions.

4.1.3.3 Modeling Abstractions

Figure 4.10 shows the meta-model elements describing the three proposed service behavior abstractions. Type FineGrainedBehavior is attached to the type BasicComponent, a component type that cannot contain further subcomponents. The CoarseGrainedBehavior is attached to type InterfaceProvidingRequiringEntity that generalizes the types System, Subsystem, CompositeComponent and BasicComponent. Type BlackBoxBehavior is attached to type InterfaceProvidingEntity, neglecting external service calls to required services. Thus, in contrast to the fine-grained abstraction level, the coarse-grained and

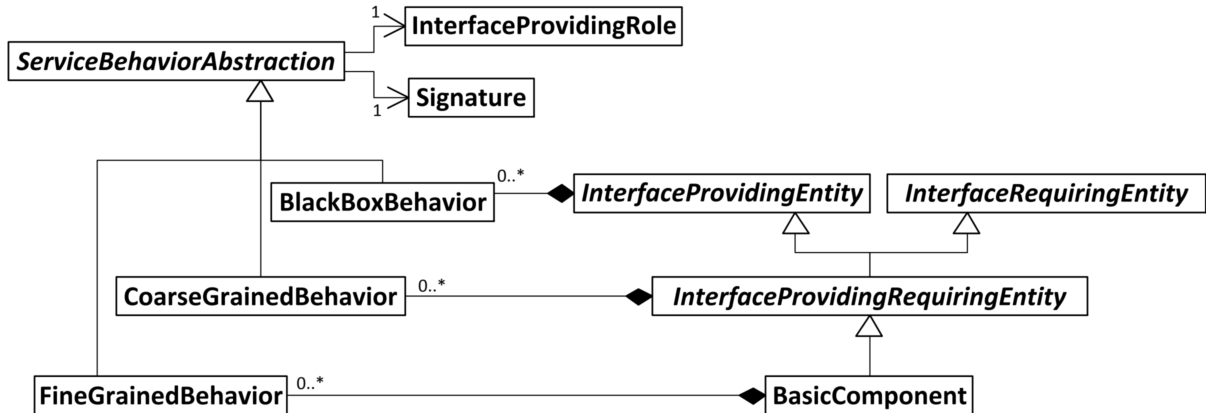


Figure 4.10: Different Service Behavior Abstractions

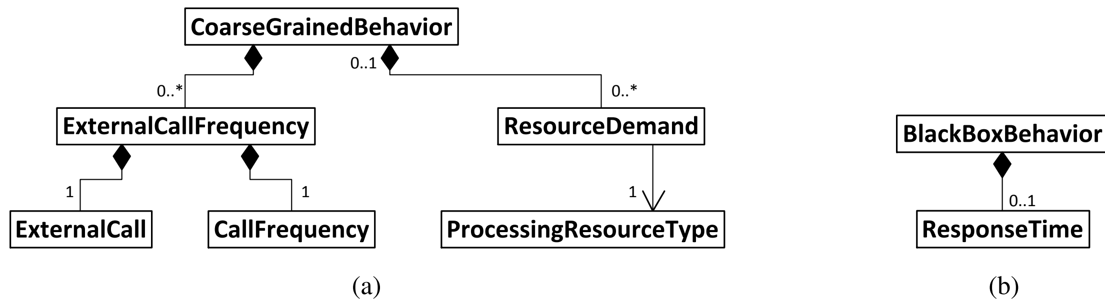


Figure 4.11: (a) Coarse-Grained and (b) Black-Box Behavior Abstractions

black-box behavior descriptions can also be attached to service-providing *composites*, i.e., ComposedStructures.

The meta-model elements for the CoarseGrainedBehavior and BlackBoxBehavior abstractions are shown in Figure 4.11. A CoarseGrainedBehavior consists of ExternalCallFrequencies and ResourceDemands. An ExternalCallFrequency characterizes the type and the number of external service calls. Type ResourceDemand captures the total service time required from a given ProcessingResourceType. A ProcessingResourceType is, e.g., a CPU, HDD or network. A BlackBoxBehavior, on the other hand, can be described with a ResponseTime characterization.

Figure 4.12 shows the meta-model elements for the fine-grained behavior abstraction. A ComponentInternalBehavior models the abstract control flow of a service implementation. Calls to required services are modeled using so-called ExternalCallActions, whereas internal computations within the component are modeled using InternalActions, characterized with ResourceDemands. Access to PassiveResources with semaphore semantics (e.g., thread pools) can be modeled via AcquireAction to obtain the resource and ReleaseAction to release the resource. Nested control flow actions like LoopAction, BranchAction or ForkAction are only used when they affect calls to required services (e.g., if a required service is called within a loop, a corresponding LoopAction is modeled; otherwise, the whole loop would be captured as part of an InternalAction). The nested control flow actions contain further ComponentInternalBehavior models, either as loop body, as forks, or as branch transitions. LoopActions and BranchActions can be characterized with loop iteration counts and branching probabilities, respectively. ForkActions can

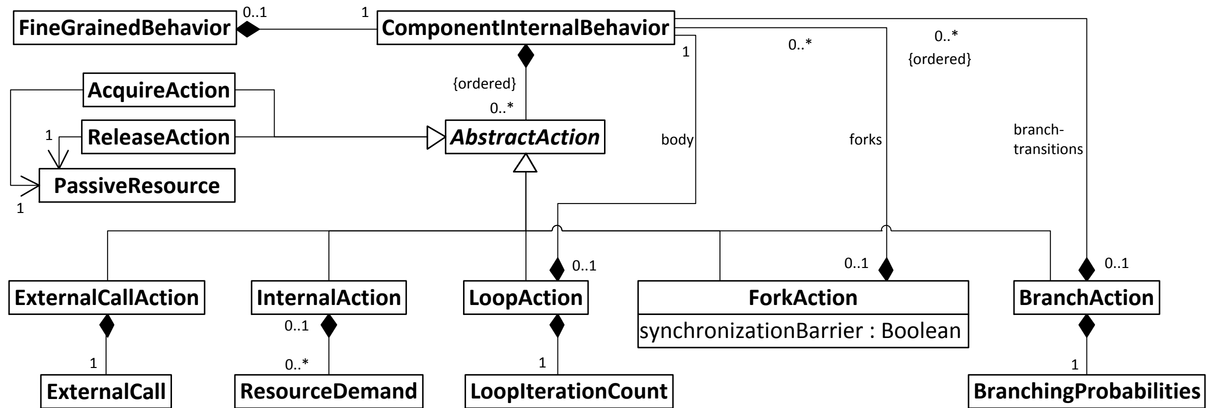


Figure 4.12: Fine-Grained Behavior Abstraction, cf. [11]

Figure 4.13: Example: *Delivery* and *ShoppingCartServlet*

be modeled either with or without a synchronization barrier. A barrier for the group of the ForkAction's forks means that the control flow only proceeds when all forks have reached the barrier.

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

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

4.1.3.4 Example

In the *WebShop* example, the *ShoppingCartServlet* provides a service called *calculateTotalCost* that calculates the cost of all items in the shopping cart including delivery costs (see Figure 4.13). To obtain the delivery costs depending on the user preferences, the *ShoppingCartServlet* either calls service *getExpressDeliveryCost* or service *getStandardDeliveryCost*. These two services are provided by the *Delivery* component. In this example, the probability of standard delivery is 0.8, and 0.2 for express delivery.

A fine-grained model of service *calculateTotalCost* is depicted in Figure 4.14. The service behavior

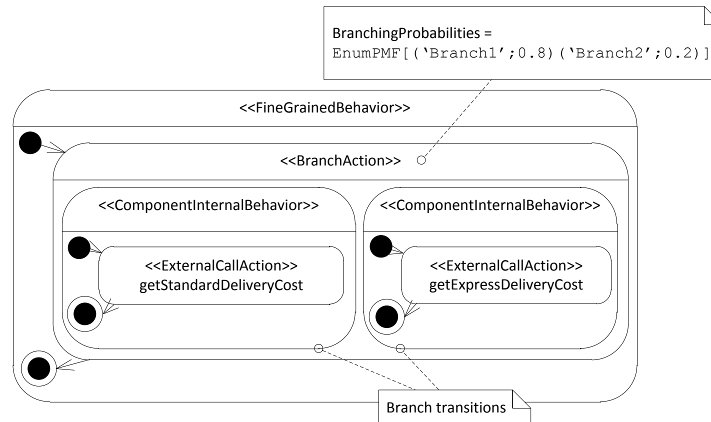


Figure 4.14: Example: Fine-Grained Behavior Abstraction of Service *calculateTotalCost* Provided by *ShoppingCartServlet*

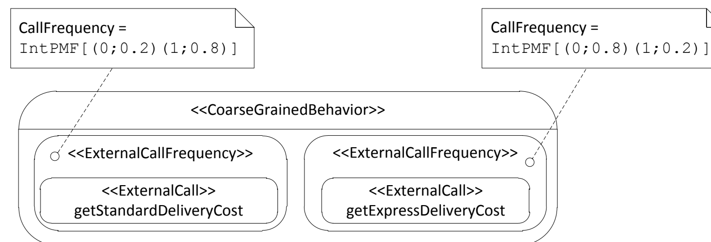


Figure 4.15: Example: Coarse-Grained Behavior Abstraction of Service *calculateTotalCost* Provided by *ShoppingCartServlet*

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

A coarse-grained model of service *calculateTotalCost* is depicted in Figure 4.15. The external service calls to *getStandardDeliveryCost* and *getExpressDeliveryCost* are modeled as they can be observed from the component boundary of component *ShoppingCartServlet*. For each call to *calculateTotalCost*, a respective external service is either called once or not at all. An external call to *getExpressDeliveryCost* has a frequency of 1 with a probability of 0.2, and 0.8 otherwise. For external call *getStandardDeliveryCost*, the probabilities are vice versa. The annotations of the form `IntPMF[(0;0.2)(1;0.8)]` are explained in Section 4.1.4. However, the exclusive relationship between the two external service calls is not reflected in the coarse-grained model. This leads to deviations when deriving the response time distribution of service *calculateTotalCost*.

4.1.4 Parameterization

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

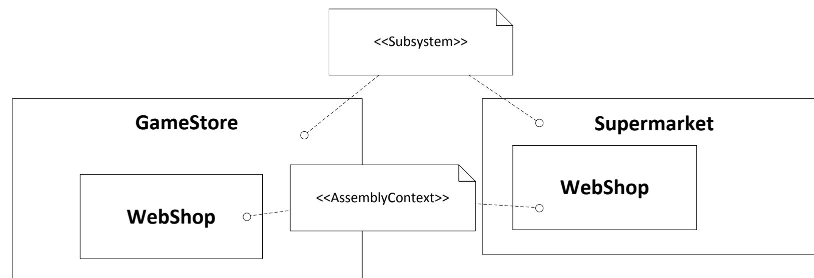


Figure 4.16: Example: WebShops for a *GameStore* and a *Supermarket*

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

4.1.4.1 Motivation

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

Assume the *WebShop* introduced in Section 4.1.2 is instantiated for different stores. For example, one instance of the web shop serves as an online supermarket and another instance of the web shop serves as a game store. See Figure 4.16 for an illustration. Technically, the component implementation is the same, but the performance behavior may differ among the two instances. One reason is the different usage behavior. While a supermarket client typically buys many items at once, a client of the game store typically buys only few items per order. Another reason is the different article database. The game store may provide video sequences when showing article details, the supermarket may only show static article images. When parameterizing a performance model, measurements of, e.g., *ShowDetailsServlet*, are likely to differ between the two shops. Although the shops use the same component types, the underlying shop data is different. In enterprise software systems, model parameters depending on the state of the database are common [50, 51]. However, it is not appropriate to model this data dependency explicitly, because it depends on internals of the database system.

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

4.1.4.2 Modeling Approach

In order to tackle different characterizations of model parameters, we provide means to specify so-called *scopes* of model parameters explicitly. A *scope* of a model parameter specifies a context where the

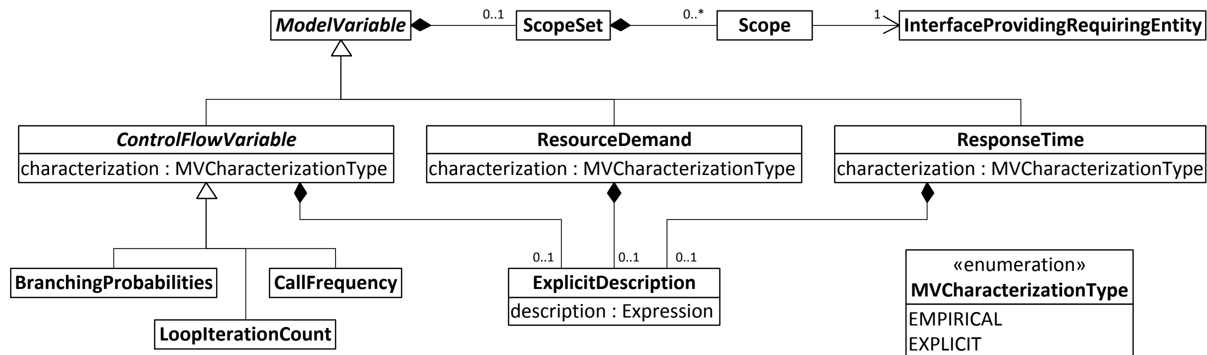


Figure 4.17: Model Variables

parameter is unique. This means, on the one hand, that measurements of the parameter can be used interchangeably among component instances provided that these instances belong to the same scope. On the other hand, it means that measurements of the parameter are not transferable across scope boundaries. Thus, if monitoring data for a given parameter is available, it should be clear based on its scope for which other instances of the component this data can be reused.

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

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

4.1.4.3 Modeling Abstractions

Figure 4.17 shows the possible ModelVariables of service behavior abstractions that have to be parameterized. On the one hand, there are timing parameters such as ResourceDemand (for coarse-grained and fine-grained service behavior abstractions) and ResponseTime (for black-box service behavior abstractions). On the other hand, there are control-flow related parameters such as CallFrequency, LoopIterationCount and BranchingProbabilities.

There are two ways to characterize the mentioned model variables. Either EMPIRICAL (default) or EXPLICIT can be chosen as a characterization type. EMPIRICAL means that the model variable has to be quantified using monitoring statistics, i.e., it is characterized using empirical data that is accessed via an interface to the monitoring infrastructure. The interface is presented in Section 4.1.6. EXPLICIT

means that the model variable is characterized explicitly. In this case, an ExplicitDescription can be used to specify a random variable by means of the Stochastic Expression (StoEx) language proposed by [52, 11]. The StoEx language allows characterizing discrete probability distributions with Probability Mass Functions (PMFs), approximating continuous probability densities with samples, or using common probability distribution functions such as the exponential distribution or the binomial distribution. Furthermore, the expression language allows specifying random variables "as a combination of several other random variables using arithmetic or boolean operations" [52, p.66].

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

The random variables of the remaining two model variables ResponseTime and ResourceDemand are typically defined on the sample space $\Omega = \mathbb{R}_{\geq 0}$, where $\omega \in \Omega$ is interpreted as timing value. They are described by a Probability Density Function (PDF) that is either approximated (see [52, p.67] for an illustration) or defined using common distributions such as the exponential distribution. An exemplary approximation $f_X^{'}$ for a density function $f_X(x)$ of a random variable X is `DoublePDF[(10.0;0.0)(30.0;0.04)(32.0;0.1)]`, meaning

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

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

As shown in Figure 4.17, a Scope is modeled as a reference to an InterfaceProvidingRequiringEntity. A ModelVariable may have several scopes, modeled as ScopeSet, because it may be assembled in different ComposedStructures. In the following, the semantics of such a ScopeSet is defined. Let v be a model variable, and c^v the (composite) component or (sub-)system where the model variable v resides. Let $S^v = \{s_1^v, \ldots, s_n^v\}$ denote the set of (composite) components or (sub-)systems referenced as Scope of v. Let $S_0^v = S^v \cup \{s_0\}$, where s_0 represents the system. An instance of c^v, as shown in Figure 4.7 in Section 4.1.1, can be identified as a sequence of AssemblyContexts $\{a_1, \ldots, a_m\}$. The container of AssemblyContext a_1 is the system, and $encapsulatedComponent(a_m) = c^v$ meaning that AssemblyContext a_m encapsulates c^v. We then define

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

Measurements of model variable v at instance $\{a_1, \ldots, a_m\}$ are then valid within $evalscope^v(\{a_1, \ldots, a_m\})$. Function $evalscope^v(\{a_1, \ldots, a_m\})$ evaluates to a (composite) component or (sub-)system instance whose

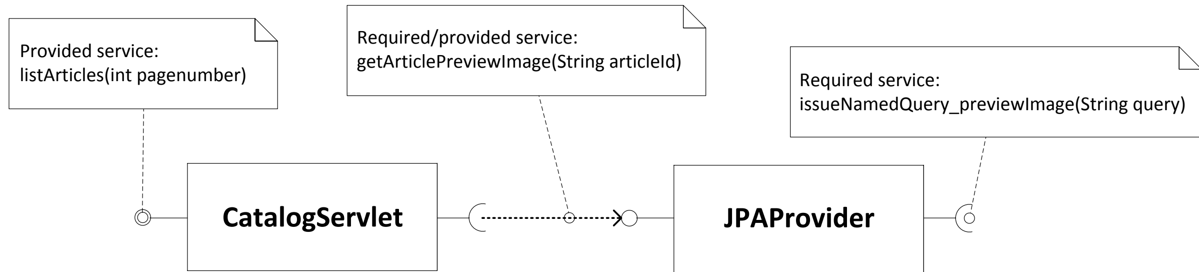


Figure 4.18: Example: *CatalogServlet* and *JPAProvider* Components

type is in the set S_0^v, namely the innermost instance when traversing from the instance identified by $\{a_1, \ldots, a_m\}$ to the system s_0. Note that the scope of a variable is only considered if EMPIRICAL is chosen as a characterization type.

4.1.4.4 Example

In the *WebShop* example, a ModelVariable v whose scope is set to the surrounding composite component *WebShop* is parameterized differently for each instance of component *WebShop*. Consequently, in Figure 4.16, the variable v has different values for the game store and the supermarket instances. If the scope is omitted, the value of variable v is shared across all component instances.

4.1.5 Probabilistic Parameter Dependencies

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

4.1.5.1 Motivation

Figure 4.18 shows how the *CatalogServlet* component in the WebShop example is connected to the *JPAProvider* component. The servlet provides a service to list the articles of the shop. The list is normally composed of multiple pages. By providing the argument *pagenumber*, the user can choose which page to view. For each article in the list, the servlet shows a preview image of the article. Preview images are loaded via the JPA provider. The JPA provider itself issues a query to load a preview image, but only if the image is not already available in the JPA cache.

We are now interested in the probability of calling *issueNamedQuery_previewImage* because it results in a costly database access. As illustrated in the fine-grained service behavior in Figure 4.19, the probability of calling the database corresponds to a branch probability in the control flow of the *getArticlePreviewImage* service. This probability depends on whether the article preview image is in the JPA cache or not. If the article (identified by parameter *articleId*) is viewed frequently, the probability of a database call is low compared to an article that is rarely shown to the users.

As discussed in Section 2.1, some architecture-level performance models for design-time analysis allow modeling dependencies of the service behavior (including branching probabilities) on input parameters passed over the service's interface upon invocation. However, in this case, the only parameter passed is *articleId*. Such a parameter does not allow modeling the dependency because the branching

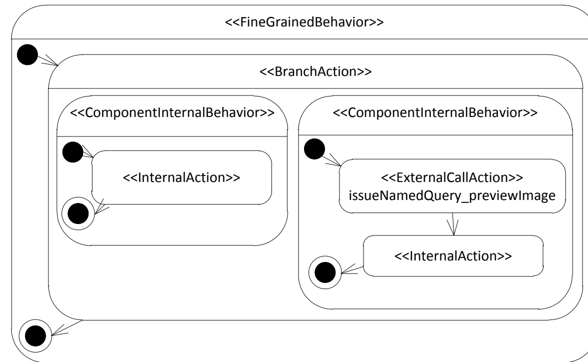


Figure 4.19: Example: Cache Miss or Cache Hit in Service *getArticlePreviewImage*

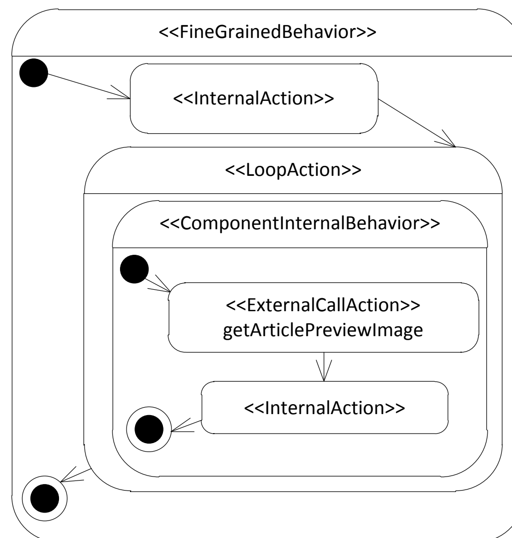


Figure 4.20: Example: Behavior of *listArticles* Service Provided by *CatalogServlet*

probabilities depend on the state of the JPA cache. Furthermore, the interface between the component and the cache is too generic to infer direct parameter dependencies. This state-dependency is typical for modern business information systems [50, 51]. The behavior of components is often dependent on the state of data containers such as caches or on persistent data stored in a database. However, modeling the state of a cache and/or a database is extremely complex and infeasible to consider as part of the performance model. Thus, in such a scenario, the approach of providing explicit characterizations of parameter dependencies is neither applicable nor appropriate.

However, in the example illustrated in Figure 4.19, there is a dependency between the branching probabilities and the service input parameter *pagenumber* of service *listArticles* provided by *CatalogServlet*. Figure 4.20 shows the fine grained behavior abstraction of the *listArticles* service. There is a loop where for each article of the requested page the corresponding preview image is requested. Intuitively, one would assume the existence of the following dependency: The higher the page number of the article list to show, the higher the probability that the articles' preview images are not in the JPA cache. This depen-

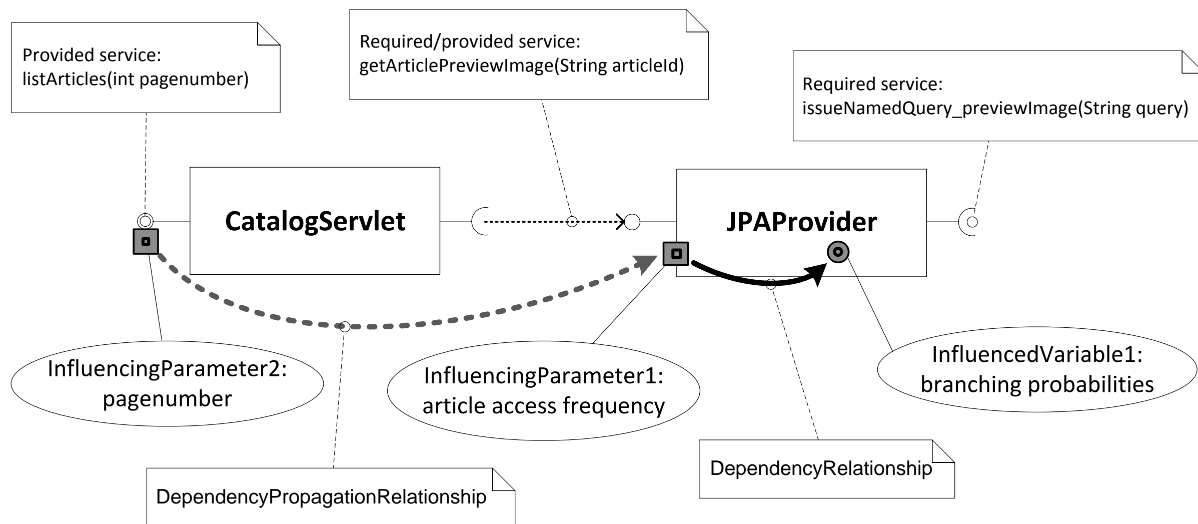


Figure 4.21: Modeling Parameter Dependencies

dency cannot be modeled using existing approaches such as PCM since, on the one hand, two separate components are involved, i.e., the *pagenumber* parameter is external to component *JPAProvider*, and on the other hand, an explicit characterization of the dependency by a function is impractical to obtain. In such a case, provided that the existence of the parameter dependency is known, monitoring statistics collected at run-time can be used to characterize the dependency probabilistically. At run-time, the dependency between the values of influencing parameter *pagenumber* and the observed relative frequency of the *issueNamedQuery_previewImage* service calls can be monitored. Using this data, the branching probabilities in Figure 4.19 can be characterized more accurately as conditional probabilities, depending on values of parameter *pagenumber* of service *listArticles*.

The parameter dependency in the example can be considered typical for enterprise software systems. The behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation [50, 51]. Such parameters are often not traceable directly over the service interface and tracing them requires looking beyond the component boundaries, e.g., the parameters might be passed to another component in the call path and/or they might be stored in a database structure queried by the invoked service. Furthermore, even if a dependency can be traced back to an input parameter of the called service, in many practical situations, providing an explicit characterization of the dependency is not feasible (e.g., using PCM's approach) and a probabilistic representation based on monitoring data is more appropriate. This situation is common in business information systems and our modeling abstractions must provide means to deal with it.

4.1.5.2 Modeling Approach

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

Figure 4.21 illustrates our modeling approach in the context of the presented example from Fig-

ure 4.18. The branching probabilities of issuing a database query or not in the *getArticlePreviewImage* service are represented as *InfluencedVariable1*. The component developer is aware of the existence of the dependency between the branch probability and the *frequency* of the article to be listed. However, the developer does not have direct access to the *pagenumber* parameter of the *listArticle* service and does not know where the parameter might be observable and traceable at run-time. Thus, to declare the existence of the dependency, the component developer first defines an *InfluencingParameter1* named *article access frequency*, representing a so-called *shadow parameter*, and provides a textual description of that parameter's semantics. Parameter *article access frequency* characterizes how frequent the article with id *articleId* is accessed. The developer can then declare a *dependency* relationship between *InfluencedVariable1* and *InfluencingParameter1*.

The developer of composite component *WebShop* is then later able to link *InfluencingParameter1* to the respective service input parameter *pagenumber* of the *CatalogServlet* component, designated as *InfluencingParameter2*. He does not explicitly know *how* the values of the page number relate to *InfluencingParameter2*, but he assumes that there is a dependency. We refer to such a link as declaration of a *dependency propagation* relationship between two influencing parameters. Having specified the influenced variable and the influencing parameters, as well as the respective dependency and dependency propagation relationships, the parameter dependency can then be characterized empirically. Our modeling approach supports both empirical and explicit characterizations for both dependency and dependency propagation relationships between model variables.

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

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

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

4.1.5.3 Modeling Abstractions

We first describe influenced variables and influencing parameters. Then we present the modeling abstractions to model the different types of relationships between model variables as they are shown in Figure 4.21. Then we describe how we characterize the introduced relationships.

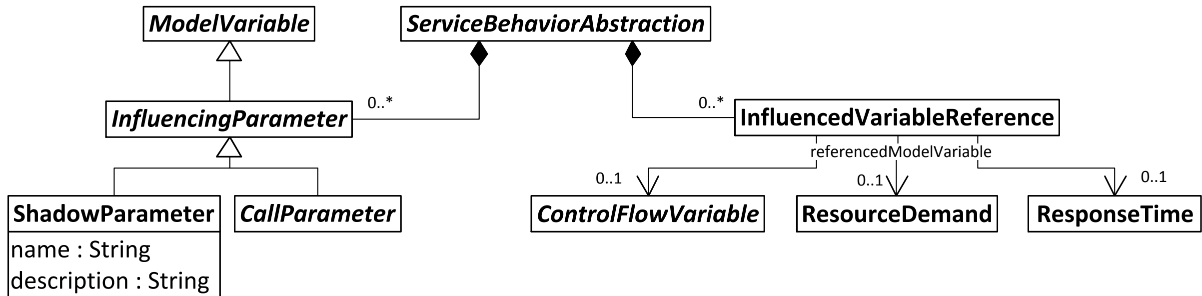


Figure 4.22: Influenced Variables and Influencing Parameters

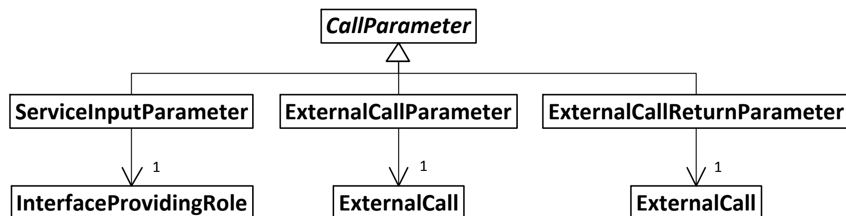


Figure 4.23: Call Parameter Hierarchy

Influenced Variables and Influencing Parameters As shown in Figure 4.22, an influenced variable is indicated by an InfluencedVariableReference, referring to either a ControlFlowVariable, a ResourceDemand, or a ResponseTime.

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

A CallParameter, see Figure 4.23, is either a service input parameter, an external call parameter, or a return parameter of an external call. Given that in performance models, a service call parameter is only modeled if it is performance-relevant (see Figure 4.24), each modeled service call parameter can be considered to have a performance influence. Furthermore, the proposed modeling abstractions support referring not only to a parameter *VALUE*, but also to other characterizations such as NUMBER_OF_ELEMENTS if the referenced data type is a collection (cf. [11]).

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

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

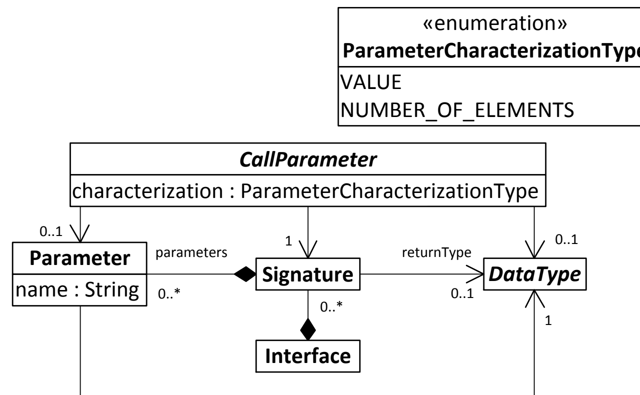


Figure 4.24: Call Parameters

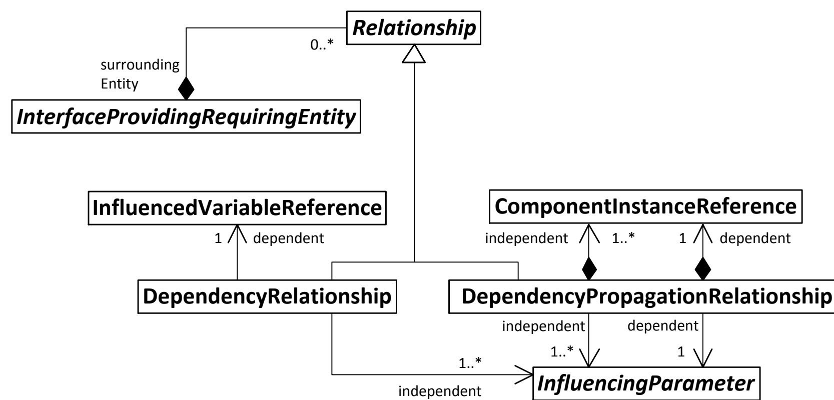


Figure 4.25: Relationships between Influenced Variables and Influencing Parameters

Relationships: Dependency and Dependency Propagation As shown in Figure 4.25, we distinguish the two types of relationships DependencyRelationship and DependencyPropagationRelationship between model variables. The former declares one influenced variable to be dependent on one or more influencing parameters (the *independent* parameters). The latter connects one or more influencing parameters (the *independent* parameters) with one influencing parameter (the *dependent* parameter)) declaring the existence of a dependency between them. Thus, an influencing parameter can play the role of a dependent parameter in one relationship, while at the same time being an independent parameter in another relationship. Both DependencyRelationship and DependencyPropagationRelationship are non-symmetric, one-directional, transitive relationships. However, note that cycles of relationships are prohibited.

A Relationship is attached to the innermost InterfaceProvidingRequiringEntity, i.e., (composite) component or (sub-)system, that surrounds the relationship. A dependency is defined at the InterfaceProvidingRequiringEntity where the influenced variable resides, and is specified by the corresponding developer. A dependency propagation is specified for a ComposedStructure that is composed of several assembly contexts. Thus, both sides of the dependency propagation need to be identified not only by the InfluencingParameters but also by ComponentInstanceReferences indicating the specific component instances where the influencing parameters reside. As depicted in Figure 4.7 in Section 4.1.1, we require the spec-

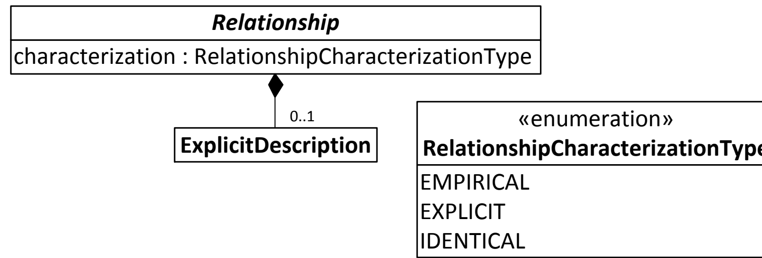


Figure 4.26: Characterization of Relationships

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

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

EMPIRICAL means that the relationship is characterized using monitoring statistics, i.e., the functional relation is determined using empirical data. The function is accessed via an interface to the monitoring infrastructure. The interface is described in Section 4.1.6.

EXPLICIT means that the relationship is characterized explicitly, i.e., with an explicit function specified using PCM's StoEx language. Such a characterization is suitable if an expression of the functional relation between the involved variables is available, the expression is modeled using model entity ExplicitDescription that corresponds to the PCM model element Expression (cf. [52, p.80]). Note that Scopes of involved ModelVariables are not considered when evaluating EXPLICIT relationships.

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

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

4.1.5.4 Example

As example for parameter dependencies, we use the parameter dependencies as they are illustrated in Figure 4.21. The service behavior abstraction of service *getArticlePreviewImage* provided by the *JPA-*

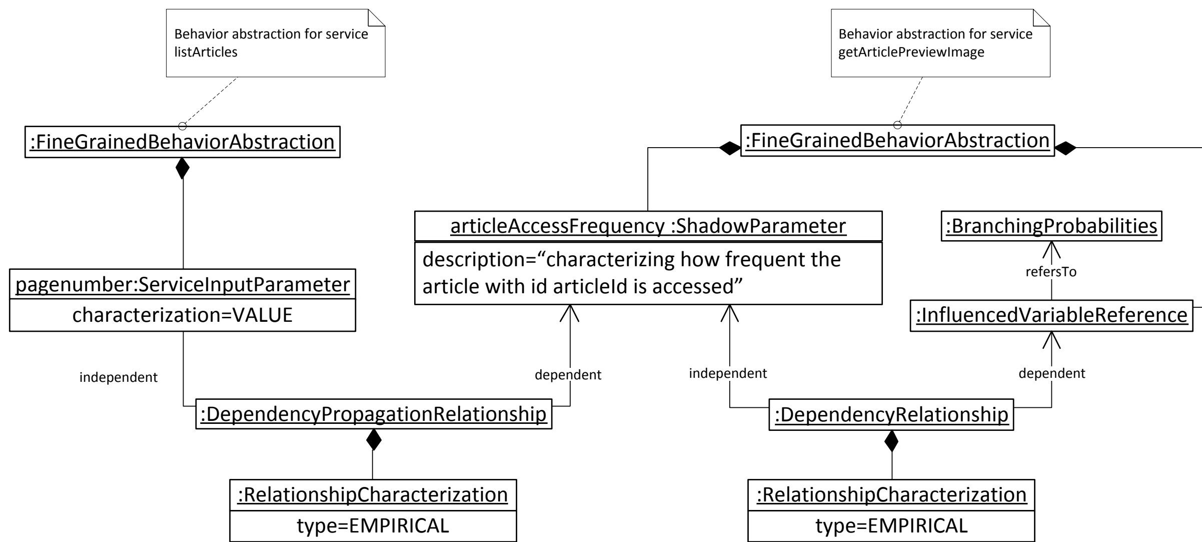


Figure 4.27: Example: Modeling Parameter Dependencies

Provider component contains an influenced variable. The influenced variable refers to the branching probabilities of the branch reflecting whether the requested data is in the cache or needs to be queried from the database (see Figure 4.19 for the branching behavior).

An excerpt of the object diagram is shown in Figure 4.27. There is a shadow parameter named *articleAccessFrequency*, also attached to the service behavior abstraction of service *getArticlePreviewImage*. The shadow parameter and the influenced variable are linked using a DependencyRelationship. The type of the RelationshipCharacterization is set to EMPIRICAL. The parameter *pagenumber* of service *listArticles*, provided by component *CatalogServlet*, is exposed as another InfluencingParameter. Then there is a DependencyPropagationRelationship with parameter *pagenumber* as independent parameter and the *articleAccessFrequency* parameter as dependent parameter. This DependencyPropagationRelationship is also characterized with type EMPIRICAL.

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

Figure 4.28 shows exemplary monitoring statistics showing how values of parameter *pagenumber* ranging from 1 to 12 relate to the probability of a cache miss. For instance, for a *pagenumber* of 4, the probability of a cache miss is monitored to be 0.23 on average. Thus, for a *pagenumber* of 4, the BranchingProbabilities can be parameterized with `EnumPMF[('Branch1';0.77)('Branch2';0.23)]` as PMF. For a *pagenumber* of 8, the probability of a cache miss is monitored to be 0.9 on average, and the BranchingProbabilities can be parameterized with `EnumPMF[('Branch1';0.1)('Branch2';0.9)]` as PMF.

4.1.6 Interface to Monitoring Infrastructure

The modeling abstractions for the application architecture level have ModelVariables and Relationships that need to be parameterized. In case the MVCharacterizationType of a ModelVariable, or the Rela-

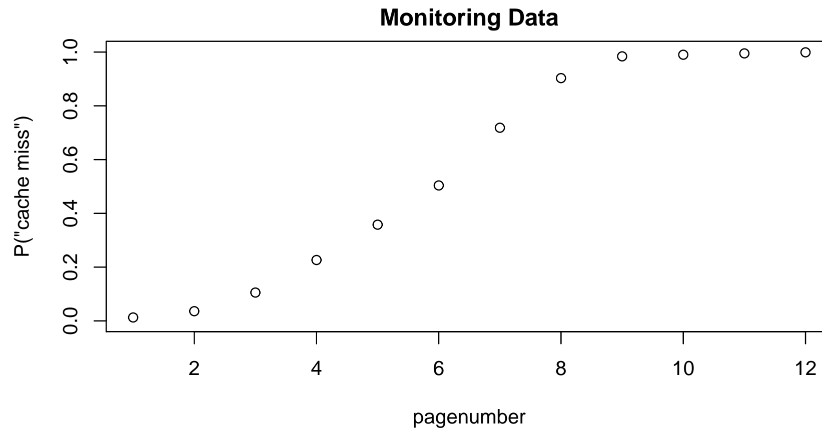


Figure 4.28: Example: Characterizing Parameter Dependencies

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

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

Listing 4.1: Monitoring Interface for the Application Level Model

tionshipCharacterizationType of a Relationship is set to EMPIRICAL, the characterization of the corresponding ModelVariable or Relationship is not specified in the application architecture model instance. Characterization type EMPIRICAL means that the values need to be obtained from monitoring statistics. Thus, the monitoring infrastructure of the running system has to be capable of providing appropriate monitoring statistics to characterize such ModelVariables and Relationships.

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

The first method *getCharacterizationForModelVariable* returns the characterization for a given ModelVariable *mv* and a given component instance where *mv* resides. The characterization is returned as a RandomVariable. For example, consider the BranchingProbabilities of the BranchAction that is part of the fine-grained behavior depicted in Figure 4.14. The behavior describes service *calculateTotalCost* that is provided by component *ShoppingCartServlet*. For one component instance of *ShoppingCart-*

Servlet, method *getCharacterizationForModelVariable* might return `EnumPMF[('Branch1';0.8) ('Branch2';0.2)]`, for a different component instance of *ShoppingCartServlet*, the method might return `EnumPMF[('Branch1';0.6)('Branch2';0.4)]`. The implementation of the method needs to consider the ScopeSet of the given ModelVariable when returning the characterization as a RandomVariable. Note that a characterization of a model variable *mv* is assumed to be available via the method *getCharacterizationForModelVariable* if *mv* is a resource demand, a response time or a control flow variable, characterized as EMPIRICAL. If the characterization is not available for such model variables, a performance prediction cannot be conducted. If *mv* is an InfluencingParameter, however, the method may return *NULL* and a performance prediction can still be conducted.

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

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

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

4.2 Resource Landscape Model

The scenario presented previously in Chapter 3 is only a small example of how today's (distributed) data centers look like. They are complex constructs of resources, interacting in different directions. On the vertical direction, resources are abstracted (e.g. by virtualization, JVM, etc.) to share them among the guests. At the same time, resources can be scaled horizontally (e.g. by adding further servers or VMs) or resources can be reassigned (e.g., by migrating virtual machines or services). For an effective system reconfiguration, it is crucial to take this information into consideration. Therefore, models that serve as a basis for reconfiguration decisions must cover diverse aspects and concepts. However, current performance models do usually not provide means to reflect such information. We will now introduce the aspects which we believe are critical for run-time performance management and effective system reconfiguration and which are conceptually presented in Section 5.2.

Resource Landscape Architecture Today's probably most general distinction of data center infrastructure on the horizontal level is the categorization of resources into computing, storage and network infrastructure. Each of this three types has its specific purpose and performance-relevant property. Each

of these types must be taken into consideration when reconfiguring the architecture of the services running in the data center. Because of their differences, each of these resources should be modeled in its own specific way. The concepts of this report will present an approach for modeling the performance-relevant properties of the computing infrastructure; storage and network infrastructure are part of other research projects (cf. [53, 54, 55, 56, 57]).

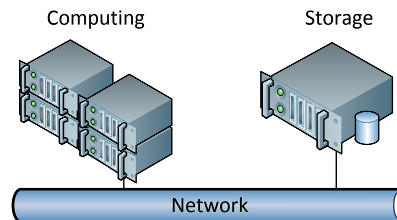


Figure 4.29: Main types of data center resources.

Another important aspect when thinking about the autonomic reconfiguration of data centers is the physical size of the data center. This has an impact on the scalability of the reconfiguration method, e.g., how many resource managers must be used. Furthermore, for the migration of services or VMs it is important to know the landscape of the data center to decide whether a migration operation is possible or not or to estimate how costly it might be.

Layers of Resources 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. Because of this flexibility, a large variety of different executing environments can be realized that all consist of similar, reoccurring elements.

Furthermore, the information about how resource containers are stacked (layering) is important for reconfiguration decisions (e.g., to decide whether an entity can be migrated or not). Another important fact is the influence of each layer on the performance. Various experiments have shown that layering the resources has influence on the performance. Therefore, the different layers must be captured in the models explicitly to predict their impact on the system's performance.

Reuse of Entities In general, the infrastructure and the software entities used in data centers are not single and unique entities. For example, a rack usually consists of the same computing infrastructure which is installed several times, virtual machines of the same type are deployed hundreds or thousands times. However, at run-time when the system is reconfigured, the configuration of a virtual machine might change. Then, this virtual machine is still of the same type as before, but with a different configuration.

With the currently available modeling abstractions such as PCM, it is necessary to model each container and its configuration explicitly. This can be very cumbersome, especially when modeling clusters of hundreds of identical machines. The intuitive idea would be to have a meta-model concept like the

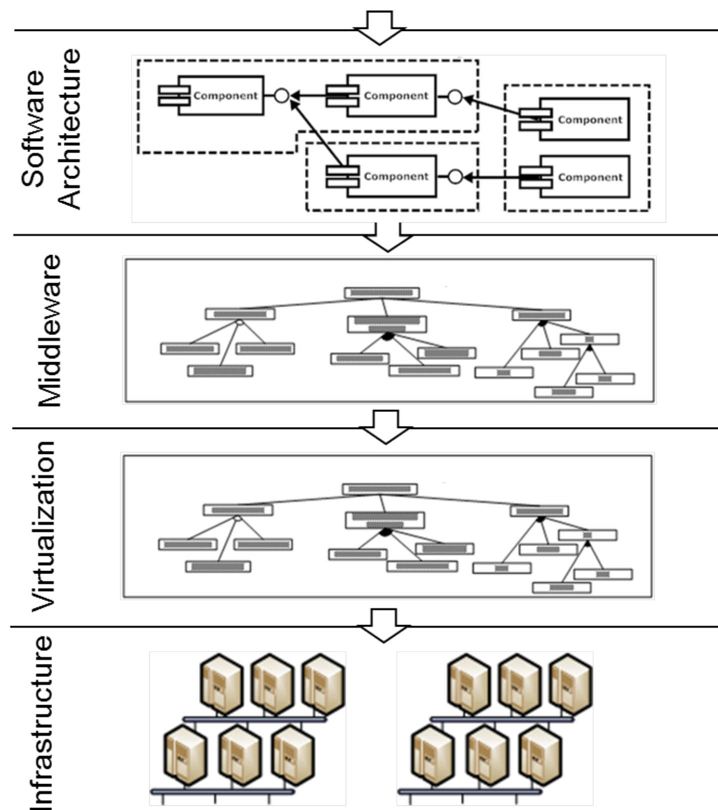


Figure 4.30: Different resource layers and their influence on the performance.

multiplicity to specify the amount of instances in the model. However, this prohibits to have individual configurations for each instance. The desired concept would support a differentiation between container types and instances of these types. The type would specify the general performance properties relevant for all instances of these types and the instance would store the performance properties of this container instance.

4.2.1 Modeling Abstractions

In the following section we present a modeling language to describe the resource landscape of distributed dynamic data centers. Instances of the resource landscape model reflect the static view of the distributed data center, i.e., they describe i) the computing infrastructure and its physical resources, and ii) the different layers within the system which also provide logical resources, e.g., virtual CPUs.

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 (cf. Section 4.2). 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.

Figure 4.31 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

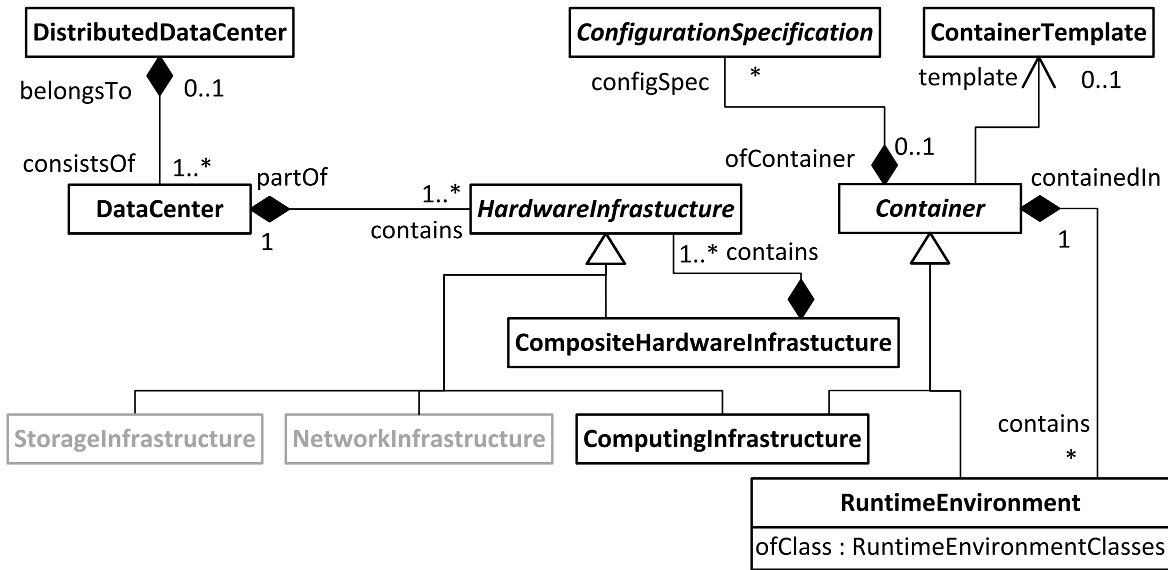


Figure 4.31: The resource landscape meta-model.

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 [53, 54, 55] and [56, 57], respectively.

4.2.1.1 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 4.31. 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.

4.2.1.2 Classes of Runtime Environments

We distinguish several general classes of runtime environments which are listed in Figure 4.32: 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 4.2: 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.

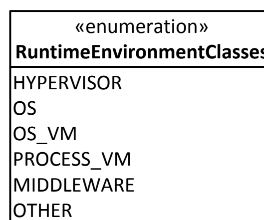


Figure 4.32: Different runtime environment classes.

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.

4.2.1.3 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 4.33.

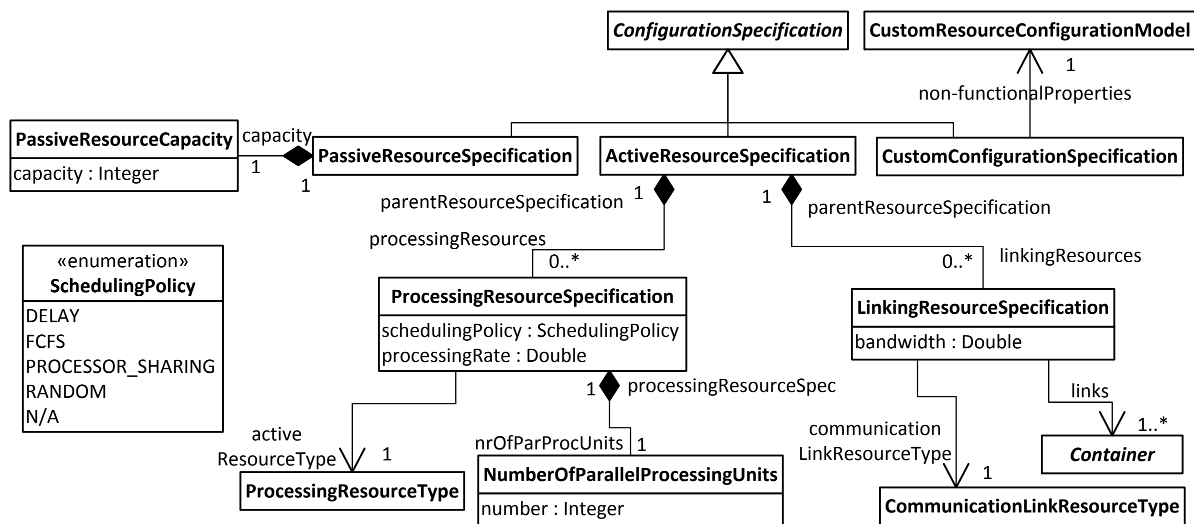


Figure 4.33: Types of resource configurations.

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 [57]. In this report, 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 [26], we presented an example CustomResourceConfigurationModel that describes the performance-relevant resource configuration for hypervisors with feature models.

4.2.1.4 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 4.34). 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, a VM of the same customer can have a similar default resource configuration specification, i.e., they are of the same type (cf. Figure 4.34). However, at run-time, we must be able to distinguish the concrete VM instances because their resource configuration specification might change.

A conceptually elegant solution to address this problem is the multilevel language engineering approach by Atkinson et al. described in [58]. This approach introduces additional levels in the Meta Object Facility (MOF) specified in [59]. 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 4.34). With Melanie [60], 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

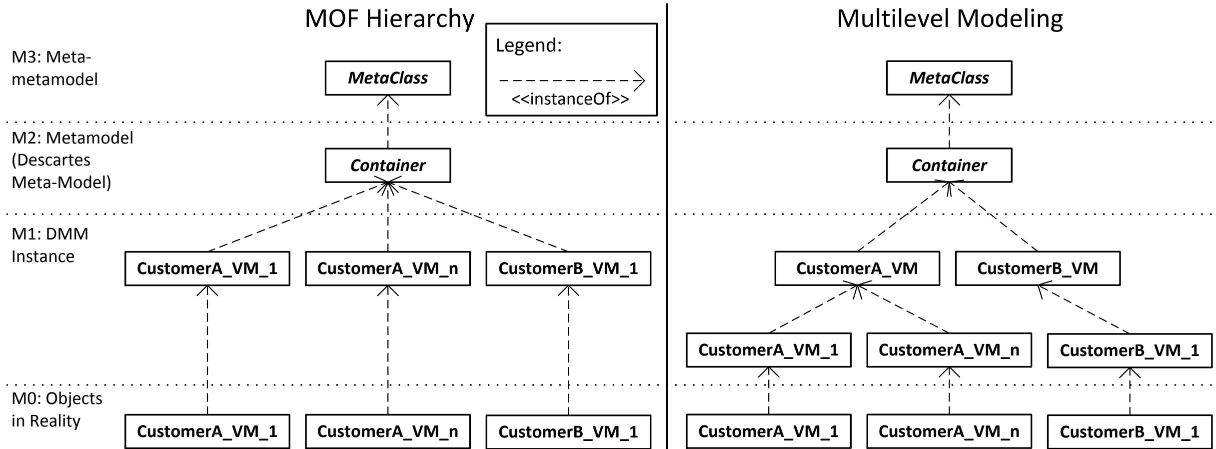


Figure 4.34: Container instances in the MOF modeling hierarchy (left) and with multilevel modeling (right).

(see Figure 4.35). 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 4.31). 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.

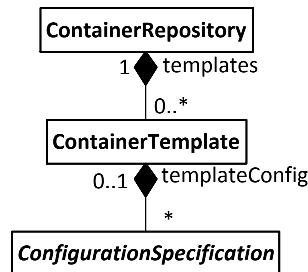


Figure 4.35: The container templates repository.

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

$R = \{r_1, r_2, \ldots, r_n\}$ 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 $RS_{VM} = \{t_1, i_2, t_3\}$

4.2.2 Example

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 4.36 depicts a resource landscape model instance in a UML-like notation, showing the hierarchy of the different resources as well as their configuration templates.

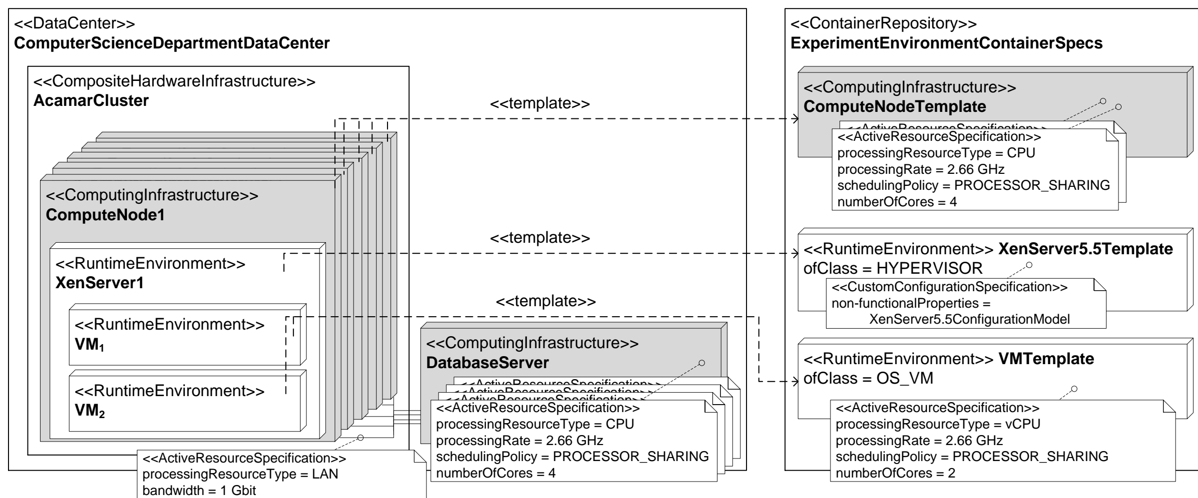


Figure 4.36: Example resource landscape model instance.

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 have been presented in [26]. 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 VCPU. It has two cores with 2.66 GHz and PROCESSOR_SHARING as scheduling policy.

4.3 Deployment Model

To capture the relationship between the resource landscape and the application architecture, one must model the relation between hardware and software.

4.3.1 Modeling Abstractions

This relation can be described with the deployment meta-model depicted in Figure 4.37. This modeling language allows to describe the allocation of software component instances of the application architecture meta-model on container instances of the resource landscape meta-model. The deployment model is based on the PCM which also models the allocation of software components to resource containers in a separate allocation context model [61]. However, because of the resource layers and the different classes of runtime environments, the interpretation of the deployment of services on resource containers is different from PCM.

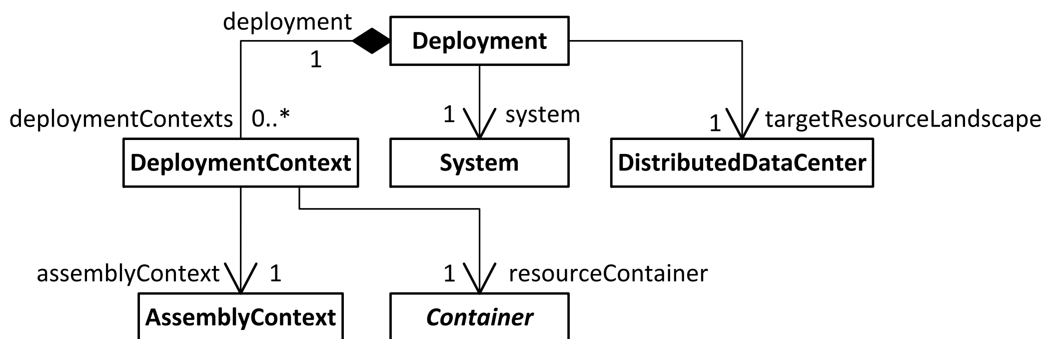


Figure 4.37: The deployment meta-model.

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 AssemblyContext. 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.

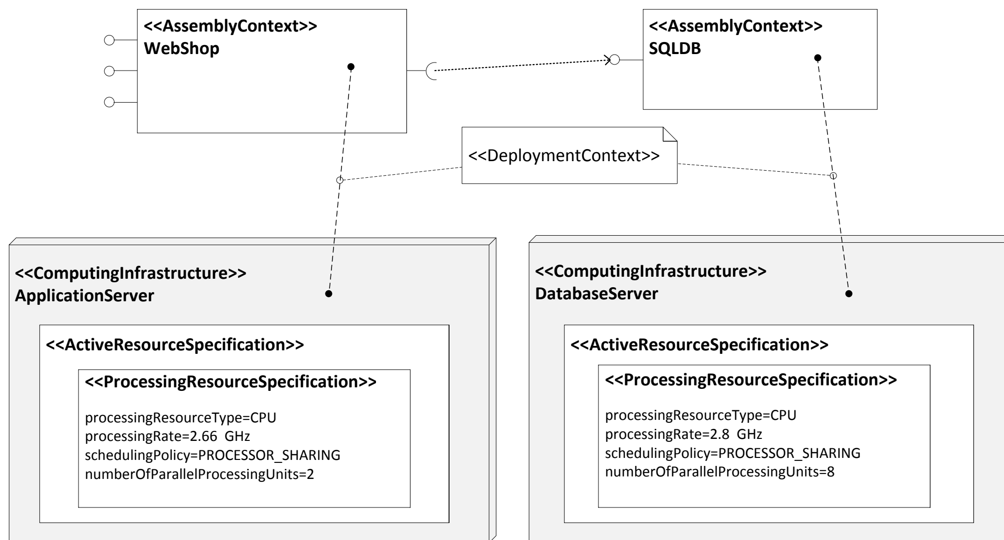


Figure 4.38: Example: WebShop Deployment

Services require different types of resources to fulfill their purpose. Hence, a constraint when deploying services to containers is that the resource types required to execute the service are actually provided by the container the service is deployed on. For example, for performance prediction, the resource demands of a service would be mapped to the resource provided by the container executing the service. In case this container is a nested container and the parent container provides a resource of the same type, the resource demand is always mapped to the subjacent resources. In each mapping step the resource demand might be adjusted according to the modeled properties of that layer or it is identically mapped in case no relevant properties are given. For example, when mapping the resource demand in a virtual machine to the hardware, the virtualization overhead can be added according to the hypervisor's performance model.

An alternative to this direct mapping of resource demands to the resources provided by the layer below is to use the more complex concept of introducing resource interfaces and controllers [62]. In this approach the resource demands can be mapped to interfaces provided by the resources. Controllers in the layers providing these interfaces take care of mapping the resource demands, e.g., adding overheads occurring in this layer.

4.3.2 Example

Figure 4.38 shows an exemplary deployment of the running example introduced in Section 4.1.2 to a simple resource landscape model instance.

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

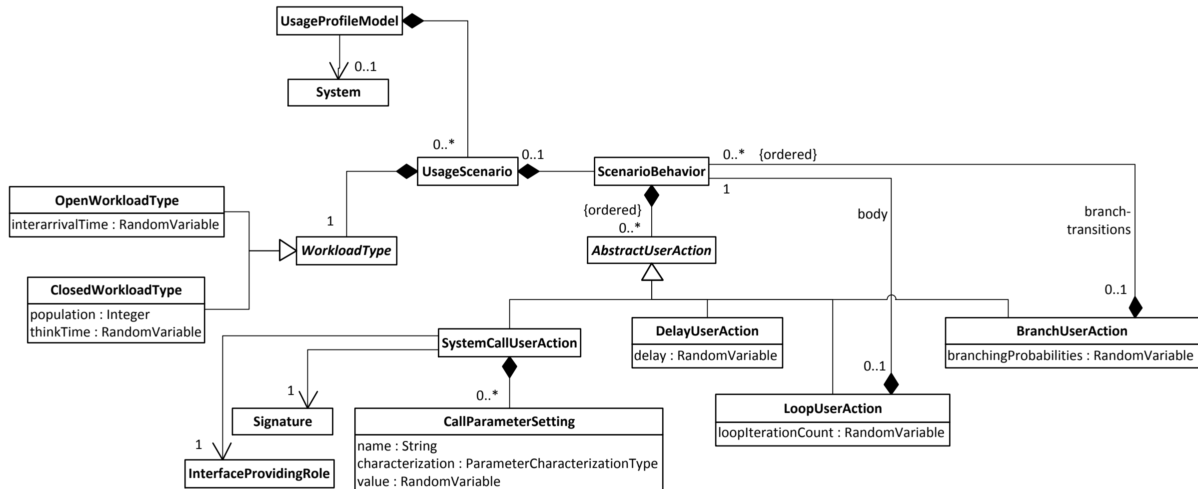


Figure 4.39: Usage Profile Model, cf. [11]

4.4 Usage Profile Model

Using the application architecture meta-model (Section 4.1) and the resource landscape and deployment meta-models (Section 4.2), an architecture-level performance model can be described. In order to conduct a performance prediction for a certain workload, the workload needs to be specified. This is done using the usage profile meta-model, also part of DML. The usage profile meta-model is based on PCM's usage model [11] and allows modeling user interactions with the system.

4.4.1 Modeling Abstractions

The modeling abstractions are shown in Figure 4.39. A UsageProfileModel consists of one or more UsageScenarios and references a System. A UsageScenario refers to a description of a WorkloadType (either an open workload or a closed workload [63]) and a ScenarioBehavior.

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

A ScenarioBehavior allows describing which services of the referenced System are called by the user using so-called SystemCallUserActions. Such a SystemCallUserAction refers to a service signature of an interface that is provided by the system. The input parameters of the service signature can (but do not have to) be set using CallParameterSettings. A CallParameterSetting consists of the name of the parameter, a parameter characterization type (see Figure 4.24) and the value of the parameter as a random variable. The SystemCallUserActions can be arranged in a control flow with delays, branches and loops. Delays, as well as branching probabilities and loop iteration counts are described with a RandomVariable.

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

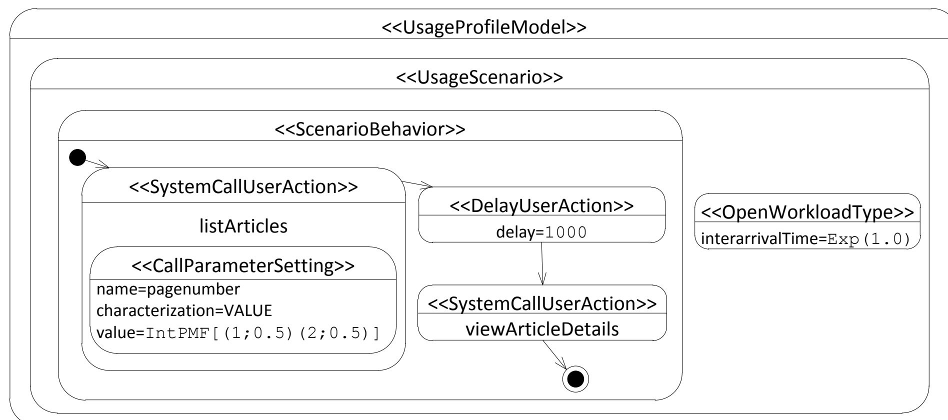


Figure 4.40: Example: Usage Profile Model Instance

4.4.2 Example

An example of a usage profile model is illustrated in Figure 4.40. It is a usage profile for the *WebShop* system presented in Section 4.1.2. It consists of one UsageScenario with an open workload. The users have an inter-arrival time of `Exp(1.0)`, i.e., an exponentially distributed inter-arrival time with mean 1.0. The user behavior starts with a call of the *listArticles* service, followed by a delay of `1000` and a service call to *viewArticleDetails*. The value of the *pagenumber* parameter of the *listArticles* service is parameterized using a PMF. With a probability of 0.5 each, either the first page or the second page is requested.

Chapter 5

Model-based System Adaptation

In this chapter, we introduce the parts of the DML relevant to i) describe the dynamic aspects of modern IT systems, infrastructures and services and to ii) model autonomic resource management at run-time. Section 5.1 explains the background and the motivation for these concepts before Section 5.2 presents the implementation.

5.1 Motivation and Background

Modern virtualized system environments are increasingly dynamic and offer high flexibility for reconfiguring systems at run-time. They usually host diverse applications of different parties and aim at utilizing resources efficiently while ensuring that Quality of Service (QoS) requirements are continuously satisfied. In such scenarios, complex adaptations to changes in the system environment are still largely performed manually by humans. Over the past decade, autonomic self-adaptation techniques aiming to minimize human intervention have become increasingly popular. However, given that adaptation processes are usually highly system specific, it is a challenge to abstract from system details enabling the reuse of adaptation strategies.

In addition to the architecture-level performance model introduced in the previous chapter, DML contains meta-models to address this flexibility and variability of modern virtualized system environments. Since current system adaptation approaches are usually based on the system-specific reconfiguration possibilities, the DML should also contain means to describe these system adaptation processes and heuristics, abstracting from the system specific details. In summary, DML provides possibilities to model system adaptation end-to-end, i.e., modeling the managed system and its adaptation possibilities up to the processes or heuristics that actually adapt the system.

To better understand how DML can be used for model-based system adaptation, we explicitly separate the DML into three parts as depicted in Figure 5.1.

System Architecture QoS Model The *system architecture QoS model* reflects the managed system from the architectural point of view. This model corresponds to the architecture-level performance model of DML introduced in Chapter 4. Thus, it comprises the same parts, such as application architecture and resource landscape (see Figure 1.2) and can be used to give a detailed description of the system architecture in a component-oriented fashion, parameterized to explicitly capture the influences of the component execution context, e.g., the workload and the hardware environment.

In this section, we use the term *system architecture QoS model* instead of architecture-level performance model because in general, other architecture-level performance models could be used, too (e.g., PCM) or even other architectural models describing other QoS attributes.

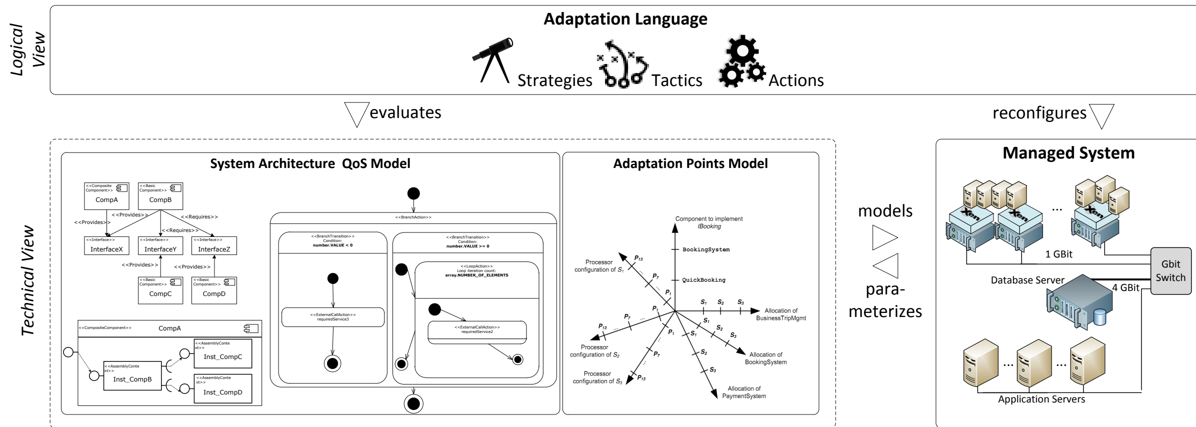


Figure 5.1: Interaction of the system, the system models and the S/T/A adaptation language.

Adaptation Points Model This model shall describe the degrees of freedom of the system architecture in the context of the system architecture QoS model, i.e., the points where the system architecture can be adapted at system run-time. Thereby, this model has to reflect the boundaries of the configuration state space, defining the possible valid configurations the system architecture can have at run-time. For example, the introduction of the resource abstraction layers like virtualization have the advantage of increased flexibility. Resources can be added/removed from VMs at run-time, VMs can be migrated while they are executed and so on. This flexibility must be reflected by the modeling abstractions. The modeling concept introduced in Chapter 4 are focused on the static aspects of the system. The adaptation points meta-model shall be used to annotate the static parts of the system architecture QoS model with their variable and hence configurable aspects. We decided to introduce these concepts in a separate model as including adaptation points within the static models has the disadvantage that the adaptation points have to be specified for each instance of the static model, too. Having a separate annotation model has the advantage that it can be used for, e.g., container types or component types. More practically, the system architecture QoS model instances exist at run-time and are then changed online according to the annotating adaptation points model. This is a difference to the approach in [47] which focuses on the meta-model level to describe which variants of model instances can be created at design time, i.e., it describes all possible variations of the component-based performance model instance. A further advantage of an annotation model is that the static and dynamic elements can be managed independently.

The adaptation points described by the adaptation points model correspond to the operations executable on the real system at run-time, e.g., adding virtual CPUs to VMs, migrating VMs or software components, or load-balancing requests. Having explicit adaptation points models is essential to decouple the knowledge of the logical aspects of the system adaptation from technical aspects. System designers can specify adaptation options based on their knowledge of the system architecture and the adaptation actions they have implemented in a manner independent of the high-level adaptation processes and plans. Furthermore, by using an explicit adaptation points model, system administrators are forced to stay within the boundaries specified by the model.

The concepts of the adaptation points model as part of the DML will be introduced in Section 5.2.

Adaptation Language To model system adaptation end-to-end, we also need a concept to describe different types of system adaptation processes. This could be deterministic and system-specific adaptation processes, e.g., rule-based system reconfiguration, or complex heuristics which adapt the system according to user-specific strategies [38]. Accordingly, the meta-model to describe system adaptation should provide enough flexibility to also model generic optimization algorithms or heuristics.

The use of an explicit adaptation points model is an important distinction of our approach from other (self-)adaptive approaches based on architecture models [39, 38]. Such approaches typically integrate the knowledge about the adaptation options and hence, the possible system states, in the operations and tactics. Also important to mention is that the system architecture QoS model is capable of reflecting much more details of the data center environment and software architecture than classical system architecture models (e.g., as used in [39]). The main resulting benefit is that we have more information about the system, thus being able to make better adaptation decisions and having more flexibility for reconfiguring the model and real system, respectively.

The concepts of this part of the DML are explained in Section 5.3.

5.2 Adaptation Points 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 DML. The aim of the adaptation points meta-model is to annotate system architecture QoS 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 Central Processing Units assigned to Virtual Machines (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 5.3.

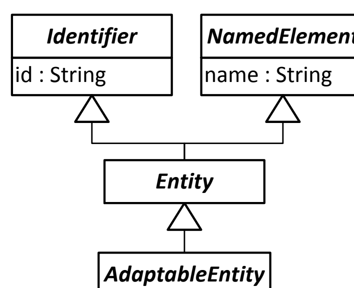


Figure 5.2: Relation of Entity and AdaptableEntity.

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 5.2) 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 [64] 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 Meta Object Facility (MOF) standard [59] 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 5.3 depicts the structure of the adaptation points meta-model. The root element AdaptationPointDescriptions collects all adaptation point annotations for adaptable elements of system architecture QoS models. We distinguish between two types of adaptations we can perform on the system architecture QoS 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.

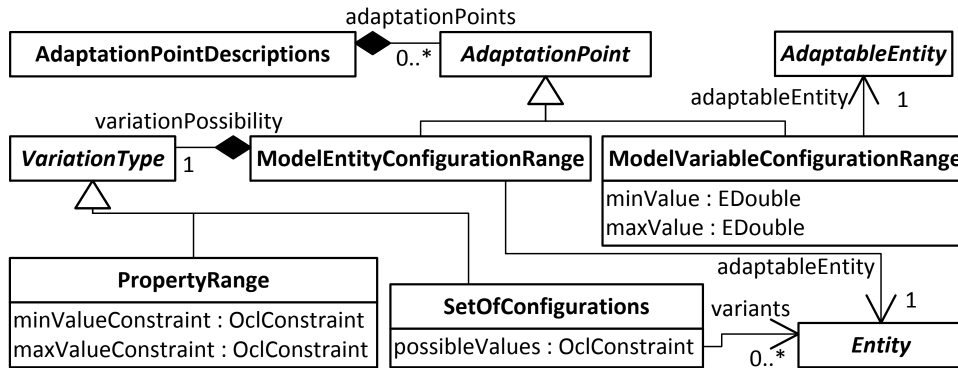


Figure 5.3: Adaptation points meta-model.

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 5.2). 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 4.33).

The ModelEntityConfigurationRange can be used to annotate other system architecture QoS 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 system architecture QoS 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 system architecture QoS 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 5.3.

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 4.2.2. Figure 5.4 depicts the example model instance. The variable resources and model entities we consider here are the number of virtual Central Processing Units (vCPUs) of a VM (*NrOfVcpus*), the number of VM instances (*VmInstances*), and the location of a VM (*VmHost*).

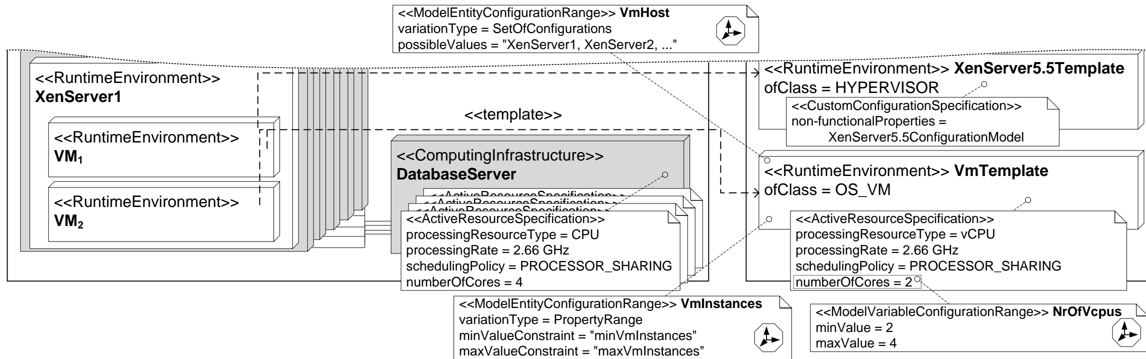


Figure 5.4: Adaptation points meta-model instance annotating the resource landscape model example in Figure 4.36.

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 Central Processing Units 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 virtual Central Processing Units (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 refers to the RuntimeEnvironment template *VmTemplate* and specifies a PropertyRange using OCL constraints (cf. Listing 5.1).

```
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 5.1: OCL constraints of *VmInstances*.

Both OCL constraints query for all Container instances and select those which refer to the same tem-

plate 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*.

5.3 Adaptation Process 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. 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 5.5). 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.

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

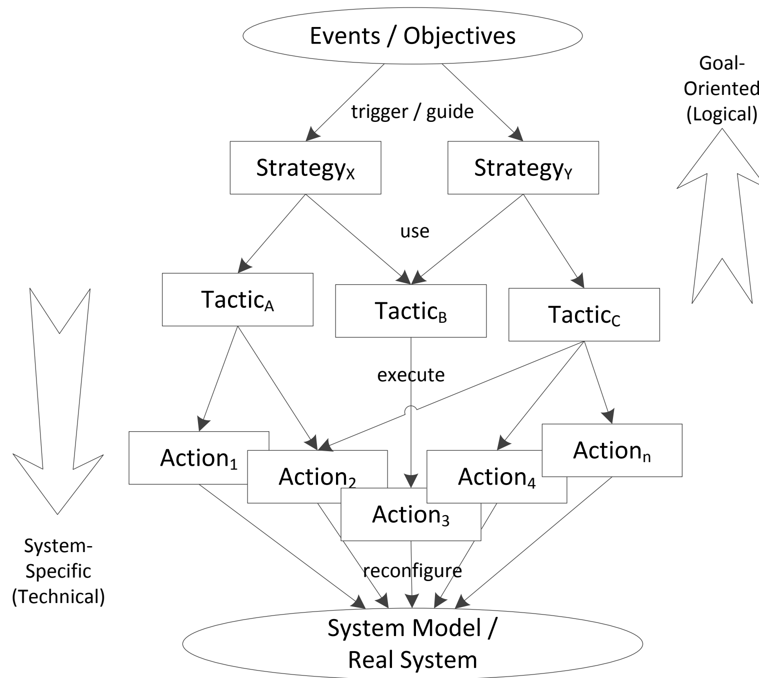


Figure 5.5: Main concepts of the adaptation process meta-model and their relations, bridging different abstraction levels.

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.

The idea of distinguishing three abstraction levels is a valid concept and can be found in other approaches, too (e.g., [40, 65]. 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 5.5).

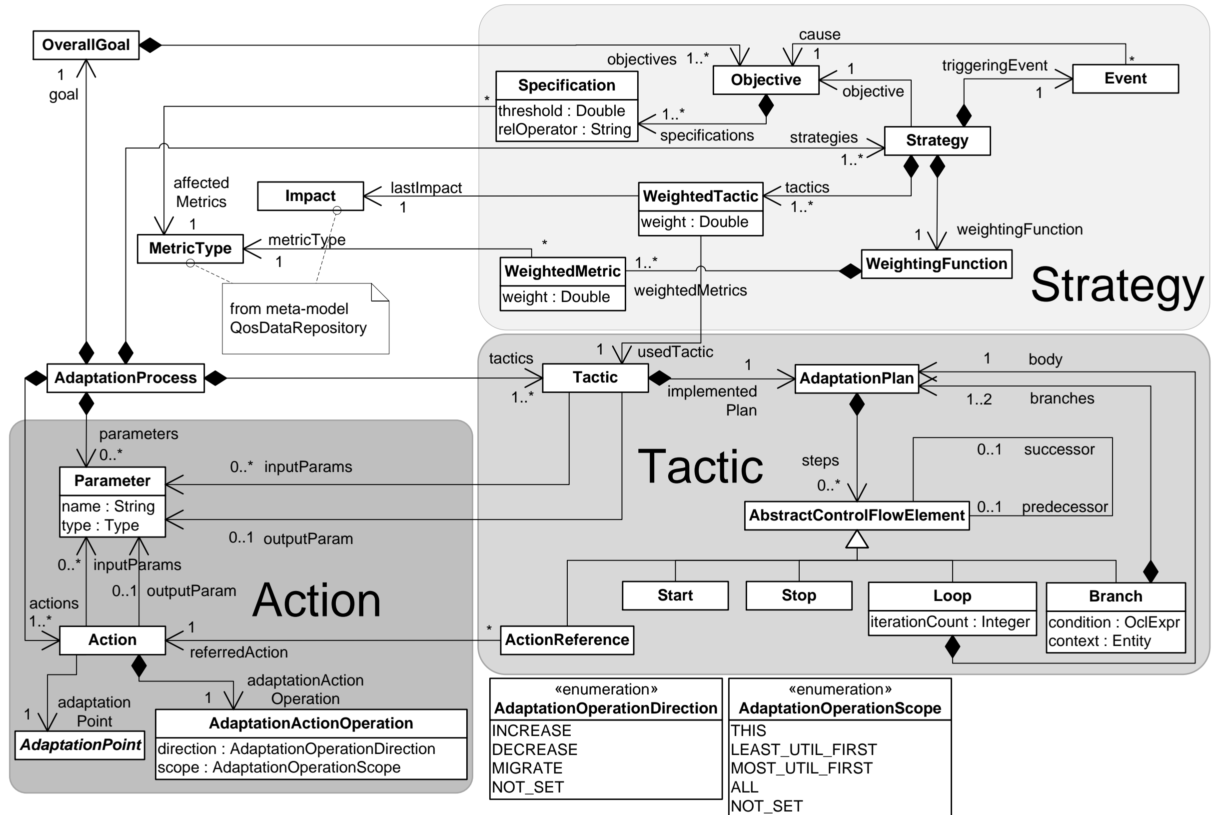


Figure 5.6: Adaptation process meta-model.

5.3.1 Actions

In Figure 5.6, 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 5.5). 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. 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 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.

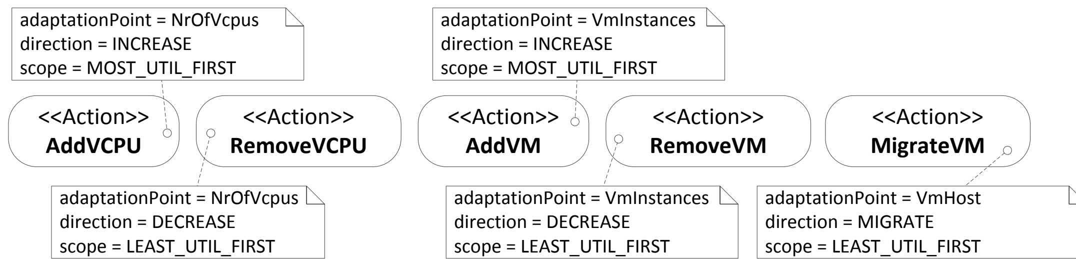


Figure 5.7: Example Actions referring to adaptation points.

Example: Figure 5.7 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 system architecture QoS model, depicted in Figure 5.4. 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, *RemoveVCPU* 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.

5.3.2 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 report, 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 5.8, 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.

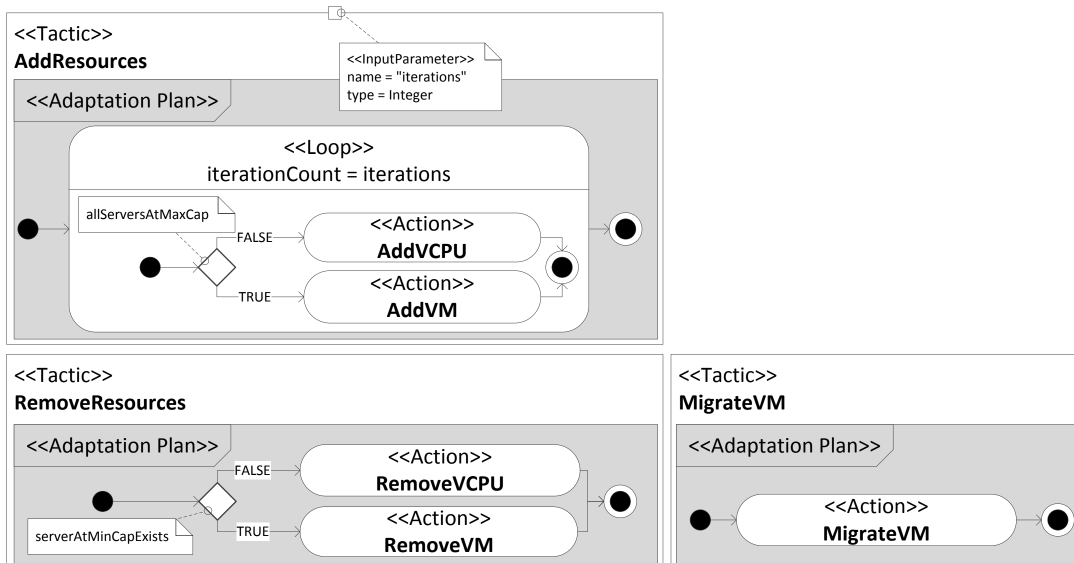


Figure 5.8: Different example Tactics using the previously specified Actions.

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 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 system architecture QoS 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 5.2. 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 as it is an implementation- and/or system-specific detail.

```
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 5.2: 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 5.3). 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 5.3: 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.

5.3.3 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 [66]. 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 5.3.5. 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 5.9 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 system architecture QoS 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.

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 system architecture QoS 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

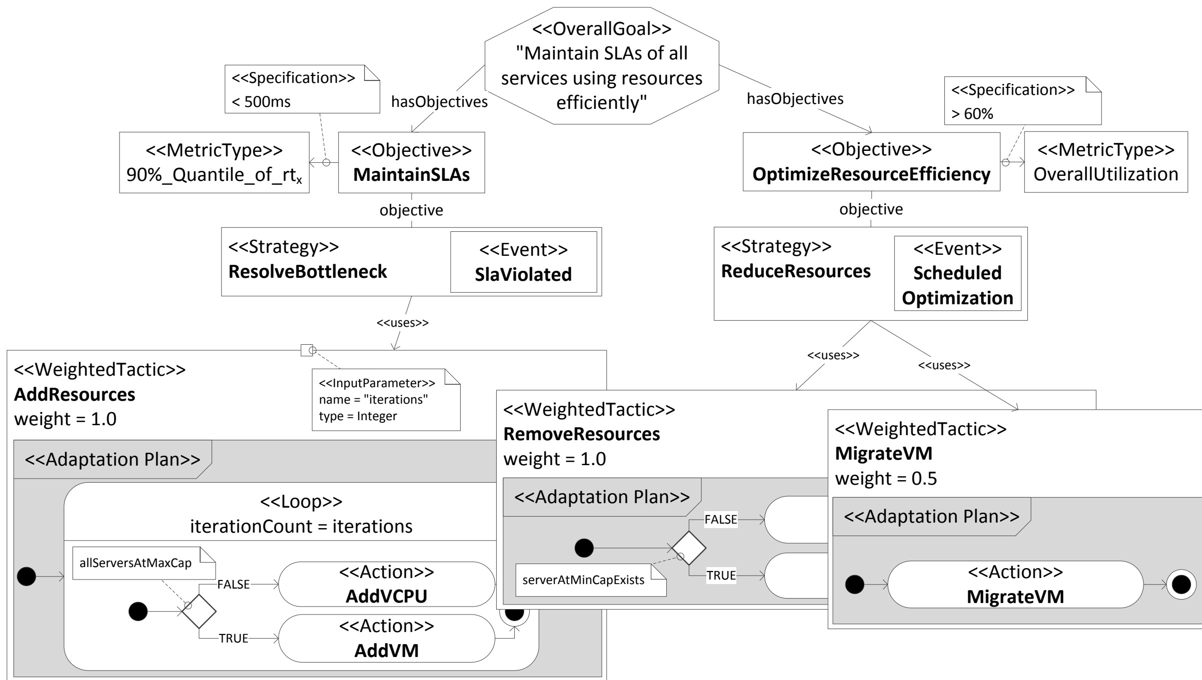


Figure 5.9: Example Strategies using Tactics with assigned weights.

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 5.3.5.

5.3.4 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 5.10). 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 $resource_n$.

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

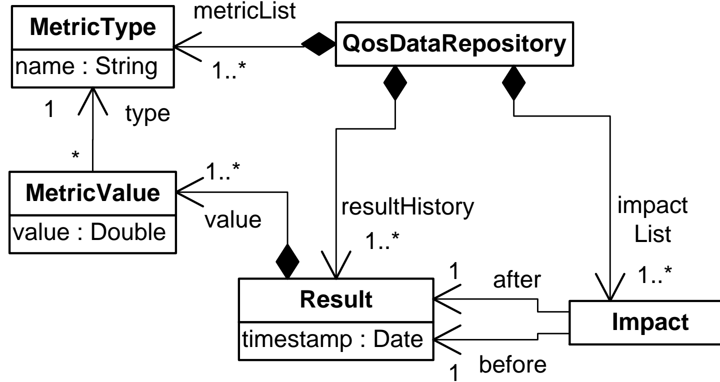


Figure 5.10: The QoS data repository meta-model.

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 [67]. Thereby, it is easier to reuse other existing tools based on SMM, e.g., the MAMBA Execution Engine and Query Language of [68]. 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 [69]. Additionally, for this meta-model we can easily provide a connector for the Query Engine [70, 71]. Then, one can use the Descartes Query Language (DQL) [70] to formulate SQL-like statements to easily query relevant performance data from our repository.

5.3.5 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

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

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

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

$V_s = \{v_s(m_1), v_s(m_2), \dots, v_s(m_m)\}$ 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. [72]) 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 5.6) 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 5.9. 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 5.9 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.

Chapter 6

Discussion

This report presented Descartes Modeling Language (DML), a new architecture-level modeling language for modeling Quality-of-Service (QoS) and resource management related aspects of modern dynamic IT systems, infrastructures and services. After providing a brief overview on related work concerning performance modeling and run-time system adaptation in Chapter 2, we introduced an exemplary online prediction scenario for DML in Chapter 3. The modeling abstractions are presented as meta-models in Chapter 4 and Chapter 5, including illustrative modeling examples.

To conclude this report, we provide a discussion of the differences between DML and Palladio Component Model (PCM) [11], with PCM being one of the most advanced architecture-level performance modeling languages in terms of parameterization [15]. Afterwards, we provide an outlook on future work.

6.1 Differences between DML and PCM

The differences between the two architecture-level performance modeling languages DML and PCM stem from their different scopes. While PCM is focussed on modeling design-time Quality of Service (QoS) properties of component-based software systems, DML focusses on the run-time aspects. As already pointed out in Section 1.2, these two different goals lead to different requirements on the modeling abstractions. In the following, we list concrete modeling aspects where PCM and DML differ:

- PCM supports and advocates the explicit specification of dependencies between model parameters, i.e., as explicit mathematical function. While this is valid in design-time scenarios, DML supports and advocates the probabilistic characterization of parameter dependencies based on monitoring data that is collected at run-time. In Section 4.1.5.1, we explained why this is more practical in run-time scenarios, and showed that explicit specifications often cannot be provided.

- DML supports to model parameter characterizations that are dependent on the component assembly, i.e., flexible characterizations for different component instances of the same component type. In PCM, parameter characterizations are fixed for the surrounding component type. Differences between component instances are intended to be captured by explicit parameterizations. Thus, in run-time scenarios where representative monitoring data is available, only DML offers a convenient approach to make use of such monitoring data for parameter characterization (see Section 4.1.4).

- PCM supports to model service behavior depending on service input parameters passed upon service invocation. However, as explained in Section 4.1.5.1, the behavior of software components is often dependent on parameters that are not available as direct service input parameters. DML

provides means to *pass* such influencing parameters to the service models whose behavior is influenced (see Section 4.1.5.2).

- In contrast to PCM, DML supports modeling multiple service behavior abstractions of different granularity for the same service. This allows for flexible performance predictions, ranging from quick bounds analysis to detailed model simulations (see Section 4.1.3).
- DML supports the modeling of complex multi-layered resource landscapes. Furthermore, it provides a template modeling mechanism that eases the re-use of resource specifications among several resource containers. This is particularly useful to model virtualization layers, to specify Virtual Machines (VMs) that stem from the same VM image (see Section 4.2).
- Furthermore, as described in Chapter 5, DML provides means to specify adaptation points as well as adaptation processes. This is out of scope for PCM.

6.2 Ongoing and Future Work

Further details on DML, e.g., on model parameterization and model solving, or on the integration of DML into an autonomic performance-aware resource management process can be found in the two phd theses [16] and [22], respectively. DML provides a basis for several areas of future work. In the following overview, we provide several pointers for research extending our work.

Load-Dependent Resource Demands In classical performance engineering, resource demands are typically assumed to be load-independent. However, modern processors implement Dynamic Voltage and Frequency Scaling (DVFS) mechanisms that adapt the processor speed depending on the current load. Thus, resource demands may appear to be load-dependent. To further increase the prediction accuracy, this load-dependency should be considered. Current versions of established model solvers are lacking support for solving performance models with load-dependent resource demands [29]. Hence, in order to support load-dependent resource demands, one should first extend the existing model solvers and then integrate the notion of a load-dependent resource demand in the model abstractions and resource demand estimation approaches.

Event-Based Systems The work in [73] describes how event-based interactions in component-based architectures can be modeled. It furthermore provides a generic approach how the developed modeling abstractions can be integrated into an architecture-level performance model. This approach can be applied to extend DML in order to add support for modeling event-based interactions such as point-to-point connections or decoupled publish/subscribe interactions. Platform-specific details about the event processing within the communication middleware are encapsulated.

Integration of Specialized Resource Modeling Approaches As part of ongoing research projects, suitable modeling abstractions for network infrastructures [56, 57, 74] and storage systems [55, 75] are under development. Given that these modeling approaches focus on network models respectively storage models, they aim to support: (i) more accurate performance analysis than what is possible with coarse-grained resource models, and (ii) further degrees-of-freedom when evaluating fine-granular configuration options of network infrastructures or storage systems. To obtain performance predictions,

these specialized performance models require detailed workload profiles as input. Using DML, such workload profiles can be derived from the modeled application layer and the corresponding usage profile. These specialized modeling approaches should be integrated in DML, on the one hand, to increase the modeling capabilities of DML, on the other hand, to simplify the applicability of the specialized models.

QoS Properties Beyond Performance The presented DML approach is focused on performance prediction, however, the general modeling approach is not limited to performance. In future work, DML could be extended to support the analysis of further QoS properties. For instance, architecture-based reliability analysis [76] could be integrated in DML in order to support evaluations of trade-offs between performance and reliability. For example, database transactions failed due to optimistic locking can be retried multiple times. This may increase reliability at the cost of performance. Other system properties such as power consumption and operating costs are gaining in importance. In particular, adding cost estimates to DML would allow multi-criteria optimizations trading-off between performance and costs (cf. [77]).

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 is of interest 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.

Self-Aware Computing Systems The long-term vision of the Descartes Research Project — the research project behind DML — is to develop new methods for the engineering of self-aware computing systems. The latter are designed with built-in online QoS prediction and self-adaptation capabilities used to enforce QoS requirements in a cost- and energy-efficient manner. For the definition of self-awareness in this context, see Section 1.4. DML lays the foundation for this vision. In the future, self-aware computing systems should be designed from the ground up with built-in self-reflective, self-predictive, and self-adaptive capabilities. Furthermore, the overall approach should be applied in industrial cooperations to showcase the applicability of our approach and thereby establish the vision of the self-aware computing paradigm.

Bibliography

[1] Samuel Kounev, Fabian Brosig, Nikolaus Huber, and Ralf Reussner, "Towards self-aware performance and resource management in modern service-oriented systems", in *Proceedings of the 7th IEEE International Conference on Services Computing (SCC 2010), July 5-10, Miami, Florida, USA*. 2010, IEEE Computer Society.

[2] Samuel Kounev, "Self-Aware Software and Systems Engineering: A Vision and Research Roadmap", in *GI Softwaretechnik-Trends, 31(4), November 2011, ISSN 0720-8928*, Karlsruhe, Germany, 2011.

[3] Dave Durkee, "Why cloud computing will never be free", *Commun. ACM*, vol. 53, no. 5, pp. 62–69, May 2010.

[4] Steve Lohr, "Amazon's trouble raises cloud computing doubts", The New York Times, April 2011, http://www.nytimes.com/2011/04/23/technology/23cloud.html?_r=1.

[5] Carl Brooks, "Cloud slas the next bugbear for enterprise it", Online, June 2011, http://searchcloudcomputing.techtarget.com/news/2240036361/Cloud-SLAs-the-next-bugbear-for-enterprise-IT.

[6] Samuel Kounev, Philipp Reinecke, Kaustubh Joshi, Jeremy Bradley, Fabian Brosig, Vlastimil Babka, S. Gilmore, and A. Stefanek, *Resilience Assessment and Evaluation of Computing Systems*, chapter Providing Dependability and Resilience in the Cloud: Challenges and Opportunities, Dagstuhl Seminar 10292. Springer Verlag, 2011.

[7] D. Menascé and V. Almeida, *Scaling for E-Business˜- Technologies, Models, Performance and Capacity Planning*, Prentice Hall, Upper Saddle River, NJ, 2000.

[8] R. Nou, S. Kounev, F. Julia, and J. Torres, "Autonomic QoS control in enterprise Grid environments using online simulation", *Journal of Systems and Software*, vol. 82, no. 3, 2009.

[9] Jim Li, John Chinneck, Murray Woodside, Marin Litoiu, and Gabriel Iszlai, "Performance model driven QoS guarantees and optimization in clouds", in *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, Washington, DC, USA, 2009, pp. 15–22, IEEE Computer Society.

[10] Gueyoung Jung, M.A. Hiltunen, K.R. Joshi, R.D. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures", in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, june 2010, pp. 62 –73.

[11] Steffen Becker, Heiko Koziolek, and Ralf Reussner, "The palladio component model for model-driven performance prediction", *Journal of Systems and Software*, vol. 82, no. 1, pp. 3 – 22, 2009.

[12] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta, "Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach", *Journal of Systems and Software*, vol. 80, no. 4, pp. 528–558, April 2007.

[13] Connie U. Smith, Catalina M. Lladó, Vittorio Cortellessa, Antinisca Di Marco, and Lloyd G. Williams, "From UML models to software performance results: an SPE process based on XML interchange formats", in *WOSP '05: Proceedings of the 5th international Workshop on Software and Performance*, New York, NY, USA, 2005, pp. 87–98, ACM Press.

[14] Object Management Group (OMG), "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)", May 2006.

[15] H. Koziolek, "Performance evaluation of component-based software systems: A survey", *Performance Evaluation*, 2009.

[16] Fabian Brosig, *Architecture-Level Software Performance Models for Online Performance Prediction*, PhD thesis, Karlsruhe Institute of Technology (KIT), 2014.

[17] Clemens Szyperski, Dominik Gruntz, and Stephan Murer, *Component Software: Beyond Object-Oriented Programming*, ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.

[18] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn, "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.*, 2014.

[19] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn, "Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning", in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*, New York, NY, USA, April 2013, pp. 187–198, ACM.

[20] VMware, "Resource Management with VMware DRS", http://www.vmware.com/pdf/vmware_drs_wp.pdf, 2006, Version 2006-06-05. Last visit: 2014-04-30.

[21] Amazon Web Services, "Amazon auto scaling", http://aws.amazon.com/documentation/autoscaling/, 2010, Last visit: 2014-03-22.

[22] Nikolaus Huber, *Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures*, PhD thesis, Karlsruhe Institute of Technology (KIT), 2014.

[23] Nikolaus Huber, André van Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev, "Modeling Run-Time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments", *Service Oriented Computing and Applications Journal (SOCA)*, 2013.

[24] Fabian Brosig, Nikolaus Huber, and Samuel Kounev, "Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems", in *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, Oread, Lawrence, Kansas, November 2011.

[25] Samuel Kounev, Konstantin Bender, Fabian Brosig, Nikolaus Huber, and Russell Okamoto, "Automated simulation-based capacity planning for enterprise data fabrics", in *4th International ICST Conference on Simulation Tools and Techniques (SIMUtools 2011), March 21–25, 2011, Barcelona, Spain*, 2011, Acceptance Rate (Full Paper): 29.8% (23/77), ICSTBest Paper Award.

[26] Nikolaus Huber, Marcel von Quast, Michael Hauck, and Samuel Kounev, "Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments", in *Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER 2011), Noordwijkerhout, The Netherlands*. May 7-9 2011, pp. 563 – 573, SciTePress.

[27] Nikolaus Huber, Fabian Brosig, and Samuel Kounev, "Model-based Self-Adaptive Resource Allocation in Virtualized Environments", in *SEAMS'11*, 2011.

[28] Philipp Meier, Samuel Kounev, and Heiko Koziolek, "Automated Transformation of Palladio Component Models to Queueing Petri Nets", in *In 19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011), Singapore, July 25–27, 2011*, July 2011.

[29] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni, "Model-Based Performance Prediction in Software Development: A Survey", *IEEE Transactions on Software Engineering*, vol. 30, no. 5, May 2004.

[30] Samuel Kounev, "Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets", *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486–502, July 2006.

[31] S. Gilmore, V. Haenel, L. Kloul, and M. Maidl, "Choreographing Security and Performance Analysis for Web Services", in *EPEW and WS-FM*, 2005, LNCS.

[32] E. Eskenazi, A. Fioukov, and D. Hammer, "Performance Prediction for Component Compositions", in *Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE7)*, 2004.

[33] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek, "Fusion: a framework for engineering self-tuning self-adaptive software systems", in *Proc. of FSE'10*. 2010, ACM.

[34] Object Management Group (OMG), "UML-SPT: UML Profile for Schedulability, Performance, and Time, v1.1", January 2005.

[35] Dorina Petriu and Murray Woodside, "An intermediate metamodel with scenarios and resources for generating performance models from uml designs", *Software and Systems Modeling (SoSyM)*, vol. 6, no. 2, pp. 163–184, June 2007.

[36] S. Kounev, F. Brosig, N. Huber, and R. Reussner, "Towards self-aware performance and resource management in modern service-oriented systems", in *Proc. of IEEE Intl Conf. on Services Computing*, 2010.

[37] Joseph L. Hellerstein, "Engineering autonomic systems", in *Proceedings of the International Conference on Autonomic Computing*, 2009.

[38] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf, "An architecture-based approach to self-adaptive software", *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May 1999.

[39] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure", *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[40] J.O. Kephart and W.E. Walsh, "An artificial intelligence perspective on autonomic computing policies", in *IEEE Int'l Workshop on Policies for Distributed Systems and Networks*, 2004.

[41] Shang-Wen Cheng, David Garlan, and Bradley Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives", in *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 2006, ACM.

[42] Shang-Wen Cheng, *Rainbow: cost-effective software architecture-based self-adaptation*, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2008, AAI3305807.

[43] N. Esfahani, S. Malek, J. Sousa, H. Gomaa, and D. Menascé, "A modeling language for activity-oriented composition of service-oriented software systems", *Model Driven Engin. Languages and Systems*, 2009.

[44] OASIS, "Web Services Business Process Execution Language Version 2.0", 2007.

[45] Ferdi Maswar, Michel R. V. Chaudron, Igor Radovanovic, and Egor Bondarev, "Improving architectural quality properties through model transformations", in *Software Engineering Research and Practice*, 2007, pp. 687–693.

[46] T. Saxena, A. Dubey, D. Balasubramanian, and G. Karsai, "Enabling self-management by using model-based design space exploration", in *IEEE EASe*, 2010.

[47] Anne Koziolek and Ralf Reussner, "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*, New York, NY, USA, June 2011, CBSE '11, pp. 103–108, ACM, New York, NY, USA.

[48] SPECjEnterprise2010 Design Document, ", http://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html. 2011.

[49] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg, "The Palladio Component Model", Tech. Rep., Karlsruhe Institute of Technology (KIT), Fakultät für Informatik, Karlsruhe, 2011.

[50] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[51] Jerome Rolia and Vidar Vetland, "Parameter estimation for performance models of distributed application systems", in *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*. 1995, CASCON '95, IBM Press.

[52] Heiko Koziolek, *Parameter Dependencies for Reusable Performance Specifications of Software Components*, PhD thesis, University of Oldenburg, Germany, March 2008.

[53] Qais Noorshams, Dominik Bruhn, Samuel Kounev, and Ralf Reussner, "Predictive Performance Modeling of Virtualized Storage Systems using Optimized Statistical Regression Techniques", in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, New York, NY, USA, 2013, ICPE'13, pp. 283–294, ACM.

[54] Qais Noorshams, Andreas Rentschler, Samuel Kounev, and Ralf Reussner, "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*, New York, NY, USA, 2013, ICPE'13, pp. 339–342, ACM.

[55] Qais Noorshams, Kiana Rostami, Samuel Kounev, Petr Tůma, and Ralf Reussner, "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*. 2013, MASCOTS'13, IEEE.

[56] Piotr Rygielski, Samuel Kounev, and Steffen Zschaler, "Model-Based Throughput Prediction in Data Center Networks", in *Proceedings of the 2nd IEEE International Workshop on Measurements and Networking (M&N 2013)*. October 2013, IEEE.

[57] Piotr Rygielski, Steffen Zschaler, and Samuel Kounev, "A Meta-Model for Performance Modeling of Dynamic Virtualized Network Infrastructures", in *International Conference on Performance Engineering (ICPE)*, 2013.

[58] C. Atkinson, M. Gutheil, and B. Kennel, "A flexible infrastructure for multilevel language engineering", *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 742–755, 2009.

[59] Object Management Group (OMG), "Meta Object Facility (MOF) Core", http://www.omg.org/spec/MOF/2.4.1/, 2011.

[60] Colin Atkinson and Ralph Gerbig, "Melanie: multi-level modeling and ontology engineering environment", in *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, New York, NY, USA, 2012, MW '12, pp. 7:1–7:2, ACM.

[61] Steffen Becker, Jens Happe, and Heiko Koziolek, "Putting Components into Context - Supporting QoS-Predictions with an explicit Context Model", in *Proceedings of the Eleventh International Workshop on Component-Oriented Programming*, July 2006.

[62] Michael Hauck, Michael Kuperberg, Klaus Krogmann, and Ralf Reussner, "Modelling Layered Component Execution Environments for Performance Prediction", in *Proceedings of the 12th International Symposium on Component Based Software Engineering (CBSE 2009)*. 2009, number 5582 in LNCS, pp. 191–208, Springer.

[63] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter, "Open versus closed: a cautionary tale", in *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, Berkeley, CA, USA, 2006, NSDI'06, USENIX Association.

[64] Anne Koziolek and Ralf Reussner, "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*, New York, NY, USA, 2011, CBSE '11, pp. 103–108, ACM.

[65] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw, "Engineering self-adaptive systems through feedback loops", in *Software Engineering for Self-Adaptive Systems*, Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, Eds., pp. 48–70. Springer-Verlag, Berlin, Heidelberg, 2009.

[66] J.O. Kephart and W.E. Walsh, "An artificial intelligence perspective on autonomic computing policies", in *International Workshop on Policies for Distributed Systems and Networks*. 2004, pp. 3–12, IEEE.

[67] Object Management Group (OMG), "Structured Metrics Meta-Model (SMM)", http://www.omg.org/spec/SMM/1.0/, 2012.

[68] Sören Frey, Andre van Hoorn, Reiner Jung, Benjamin Kiel, and Wilhelm Hasselbring, "MAMBA: Model-Based Software Analysis Utilizing OMG's SMM", in *Proceedings of the 14. Workshop Software-Reengineering (WSR '12)*, 2012, pp. 37–38, Also appeared in Softwaretechnik-Trends 32(2) (May 2012) 49-50.

[69] André van Hoorn, Jan Waller, and Wilhelm Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis", in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Apr. 2012, pp. 247–248, ACM.

[70] Fabian Gorsler, Fabian Brosig, and Samuel Kounev, "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, 2014, ACM.

[71] Fabian Gorsler, "Online Performance Queries for Architecture-Level Performance Models", Master's thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, July 2013.

[72] Christian Blum and Andrea Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison", *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, Sept. 2003.

[73] Christoph Rathfelder, *Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation*, vol. 10 of *The Karlsruhe Series on Software Design and Quality*, KIT Scientific Publishing, Karlsruhe, Germany, 2013.

[74] Piotr Rygielski and Samuel Kounev, "Data Center Network Throughput Analysis using Queueing Petri Nets", in *34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Wokrshops). 4th International Workshop on Data Center Performance, (DCPerf 2014)*, 2014.

[75] Qais Noorshams, Roland Reeb, Andreas Rentschler, Samuel Kounev, and Ralf Reussner, "Enriching Software Architecture Models with Statistical Models for Performance Prediction in Modern Storage Environments", in *Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*, 2014, CBSE '14.

[76] Franz Brosch, Heiko Koziolek, Barbora Buhnova, and Ralf Reussner, "Architecture-based reliability prediction with the palladio component model", *Transactions on Software Engineering*, vol. 38, no. 6, 2011.

[77] Anne Koziolek, Danilo Ardagna, and Raffaela Mirandola, "Hybrid multi-attribute QoS optimization in component based software systems", *Journal of Systems and Software*, vol. 86, no. 10, 2013.

List of Acronyms and Abbreviations

CPU	Central Processing Unit.
DML	Descartes Modeling Language.
DQL	Descartes Query Language.
DVFS	Dynamic Voltage and Frequency Scaling.
HDD	Hard Disk Drive.
JPA	Java Persistence API.
JVM	Java Virtual Machine.
MOF	Meta Object Facility.
OCL	Object Constraint Language.
PCM	Palladio Component Model.
PDF	Probability Density Function.
PMF	Probability Mass Function.
QoS	Quality of Service.
RDSEFF	Resource Demanding Service Effect Specification.
S/T/A	Strategies/Tactics/Actions.
SLA	Service Level Agreement.
SMM	Structured Metrics Metamodel.
StoEx	Stochastic Expression.
UML	Unified Modeling Language.
vCPU	virtual Central Processing Unit.
VM	Virtual Machine.